Intel® 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

March 2025

Notice: The Intel® 64 and IA-32 architectures may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in the specification updates.

Document Number: 252046-079

Contents

Revision History

Revision	Description	Date
-001	• Initial release	November 2002
-002	• Added 1-10 Documentation Changes. • Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual	December 2002
-003	• Added 9 -17 Documentation Changes. • Removed Documentation Change #6 - References to bits Gen and Len Deleted. • Removed Documentation Change #4 - VIF Information Added to CLI Discussion	February 2003
-004	• Removed Documentation changes 1-17. • Added Documentation changes 1-24.	June 2003
-005	• Removed Documentation Changes 1-24. • Added Documentation Changes 1-15.	September 2003
-006	• Added Documentation Changes 16- 34.	November 2003
-007	• Updated Documentation changes 14, 16, 17, and 28. • Added Documentation Changes 35-45.	January 2004
-008	• Removed Documentation Changes 1-45. • Added Documentation Changes 1-5.	March 2004
-009	• Added Documentation Changes 7-27.	May 2004
-010	• Removed Documentation Changes 1-27. • Added Documentation Changes 1.	August 2004
-011	• Added Documentation Changes 2-28.	November 2004
-012	• Removed Documentation Changes 1-28. • Added Documentation Changes 1-16.	March 2005
-013	• Updated title. • There are no Documentation Changes for this revision of the document.	July 2005
-014	• Added Documentation Changes 1-21.	September 2005
-015	• Removed Documentation Changes 1-21. • Added Documentation Changes 1-20.	March 9, 2006
-016	• Added Documentation changes 21-23.	March 27, 2006
-017	• Removed Documentation Changes 1-23. • Added Documentation Changes 1-36.	September 2006
-018	• Added Documentation Changes 37-42.	October 2006
-019	• Removed Documentation Changes 1-42. • Added Documentation Changes 1-19.	March 2007
-020	• Added Documentation Changes 20-27.	May 2007
-021	• Removed Documentation Changes 1-27. • Added Documentation Changes 1-6	November 2007
-022	• Removed Documentation Changes 1-6 • Added Documentation Changes 1-6	August 2008
-023	• Removed Documentation Changes 1-6 • Added Documentation Changes 1-21	March 2009

Revision	Description	Date
-024	• Removed Documentation Changes 1-21 • Added Documentation Changes 1-16	June 2009
-025	• Removed Documentation Changes 1-16 • Added Documentation Changes 1-18	September 2009
-026	• Removed Documentation Changes 1-18 • Added Documentation Changes 1-15	December 2009
-027	• Removed Documentation Changes 1-15 • Added Documentation Changes 1-24	March 2010
-028	• Removed Documentation Changes 1-24 • Added Documentation Changes 1-29	June 2010
-029	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	September 2010
-030	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	January 2011
-031	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	April 2011
-032	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-14	May 2011
-033	• Removed Documentation Changes 1-14 • Added Documentation Changes 1-38	October 2011
-034	• Removed Documentation Changes 1-38 • Added Documentation Changes 1-16	December 2011
-035	• Removed Documentation Changes 1-16 • Added Documentation Changes 1-18	March 2012
-036	• Removed Documentation Changes 1-18 • Added Documentation Changes 1-17	May 2012
-037	• Removed Documentation Changes 1-17 • Added Documentation Changes 1-28	August 2012
-038	• Removed Documentation Changes 1-28 • Add Documentation Changes 1-22	January 2013
-039	• Removed Documentation Changes 1-22 • Add Documentation Changes 1-17	June 2013
-040	• Removed Documentation Changes 1-17 • Add Documentation Changes 1-24	September 2013
-041	• Removed Documentation Changes 1-24 • Add Documentation Changes 1-20	February 2014
-042	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-8	February 2014
-043	• Removed Documentation Changes 1-8 • Add Documentation Changes 1-43	June 2014
-044	• Removed Documentation Changes 1-43 • Add Documentation Changes 1-12	September 2014
-045	• Removed Documentation Changes 1-12 • Add Documentation Changes 1-22	January 2015
-046	• Removed Documentation Changes 1-22 • Add Documentation Changes 1-25	April 2015
-047	• Removed Documentation Changes 1-25 • Add Documentation Changes 1-19	June 2015

Revision	Description	Date
-048	• Removed Documentation Changes 1-19 • Add Documentation Changes 1-33	September 2015
-049	• Removed Documentation Changes 1-33 • Add Documentation Changes 1-33	December 2015
-050	• Removed Documentation Changes 1-33 • Add Documentation Changes 1-9	April 2016
-051	• Removed Documentation Changes 1-9 • Add Documentation Changes 1-20	June 2016
-052	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-22	September 2016
-053	• Removed Documentation Changes 1-22 • Add Documentation Changes 1-26	December 2016
-054	• Removed Documentation Changes 1-26 • Add Documentation Changes 1-20	March 2017
-055	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-28	July 2017
-056	• Removed Documentation Changes 1-28 • Add Documentation Changes 1-18	October 2017
-057	• Removed Documentation Changes 1-18 • Add Documentation Changes 1-29	December 2017
-058	• Removed Documentation Changes 1-29 • Add Documentation Changes 1-17	March 2018
-059	• Removed Documentation Changes 1-17 • Add Documentation Changes 1-24	May 2018
-060	• Removed Documentation Changes 1-24 • Add Documentation Changes 1-23	November 2018
-061	• Removed Documentation Changes 1-23 • Add Documentation Changes 1-21	January 2019
-062	• Removed Documentation Changes 1-21 • Add Documentation Changes 1-28	May 2019
-063	• Removed Documentation Changes 1-28 • Add Documentation Changes 1-34	October 2019
-064	• Removed Documentation Changes 1-34 • Add Documentation Changes 1-36	May 2020
-065	• Removed Documentation Changes 1-36 • Add Documentation Changes 1-31	November 2020
-066	• Removed Documentation Changes 1-31 • Add Documentation Changes 1-24	April 2021
-067	• Removed Documentation Changes 1-24 • Add Documentation Changes 1-30	June 2021
-068	• Removed Documentation Changes 1-30 • Add Documentation Changes 1-29	December 2021
-069	• Removed Documentation Changes 1-29 • Add Documentation Changes 1-18	April 2022
-070	• Removed Documentation Changes 1-18 • Add Documentation Changes 1-41	December 2022
-071	• Removed Documentation Changes 1-41 • Add Documentation Changes 1-23	March 2023

Revision	Description	Date
-072	• Removed Documentation Changes 1-23 • Add Documentation Changes 1-19	June 2023
-073	• Removed Documentation Changes 1-19 • Add Documentation Changes 1-19	September 2023
-074	• Removed Documentation Changes 1-19 • Add Documentation Changes 1-20	December 2023
-075	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-20	March 2024
-076	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-8	June 2024
-077	• Removed Documentation Changes 1-8 • Add Documentation Changes 1-27	October 2024
-078	• Removed Documentation Changes 1-27 • Add Documentation Changes 1-15	December 2024
-079	• Removed Documentation Changes 1-15 • Add Documentation Changes 1-17	March 2025

§

Preface

This document is an update to the specifications contained in the Affected Documents table below. This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Affected Documents

Document Title	Document Number/ Location
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture	253665
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L	253666
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-U	253667
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V	326018
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference, W-Z	334569
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1	253668
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2	253669
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3	326019
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4	332831
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model Specific Registers	335592

Nomenclature

Documentation Changes include typos, errors, or omissions from the current published specifications. These will be incorporated in any new release of the specification.

Summary Tables of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 architectures. This table uses the following notations:

Codes Used in Summary Tables

A violet change bar to left of table row indicates this erratum is either new or modified from the previous version of the document.

Documentation Changes

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 4, Volume 1
2	Updates to Chapter 5, Volume 1
3	Updates to Chapter 16, Volume 1
4	Updates to Appendix A, Volume 1
5	Updates to Chapter 2, Volume 2A
6	Updates to Chapter 3, Volume 2A
7	Updates to Chapter 4, Volume 2B
8	Updates to Chapter 5, Volume 2C
9	Updates to Chapter 6, Volume 2D
10	Updates to Chapter 17, Volume 3B
11	Updates to Chapter 21, Volume 3B
12	Updates to Chapter 26, Volume 3C
13	Updates to Chapter 27, Volume 3C
14	Updates to Chapter 29, Volume 3C
15	Updates to Chapter 39, Volume 3D
16	Updates Appendix B, Volume 3D
17	Updates to Chapter 2, Volume 4

Documentation Changes

Changes to the Intel® 64 and IA-32 Architectures Software Developer's Manual volumes follow, and are listed by chapter. Only chapters with changes are included in this document.

1. Updates to Chapter 4, Volume 1

Change bars and violet text show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

Changes to this chapter:

- Added Section 4.2.3, "Brain Float16."

CHAPTER 4
DATA TYPES

This chapter introduces data types defined for the Intel 64 and IA-32 architectures. A section at the end of this chapter describes the real-number and floating-point concepts used in x87 FPU and Intel SSE, SSE2, SSE3, SSSE3, SSE4, and AVX extensions.

4.1 FUNDAMENTAL DATA TYPES

The fundamental data types are bytes, words, doublewords, quadwords, and double quadwords (see Figure 4-1). A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), a quadword is 8 bytes (64 bits), and a double quadword is 16 bytes (128 bits). A subset of the IA-32 architecture instructions operates on these fundamental data types without any additional operand typing.

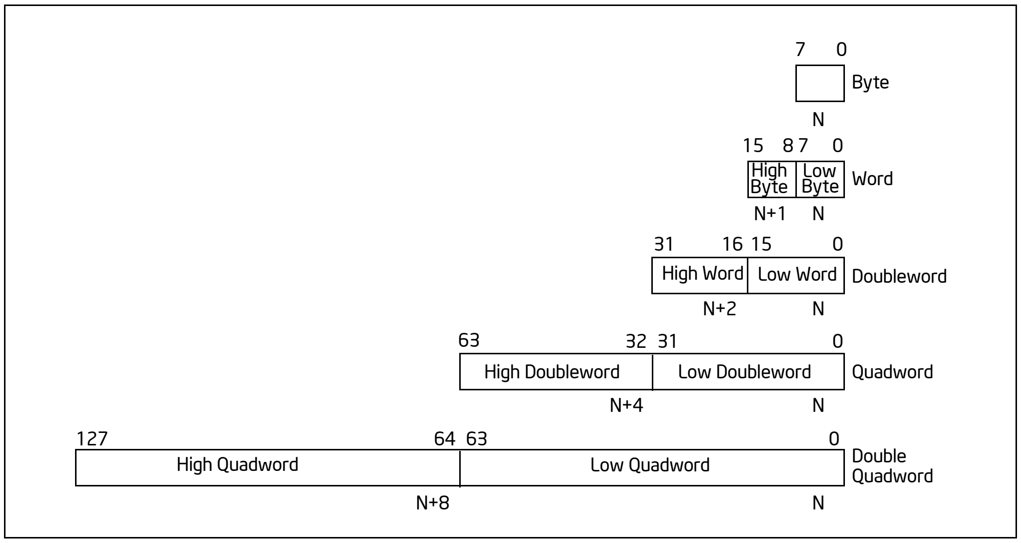


Figure 4-1. Fundamental Data Types

The quadword data type was introduced into the IA-32 architecture in the Intel486 processor; the double quadword data type was introduced in the Pentium III processor with the Intel SSE extensions.

Figure 4-2 shows the byte order of each of the fundamental data types when referenced as operands in memory. The low byte (bits 0 through 7) of each data type occupies the lowest address in memory and that address is also the address of the operand.

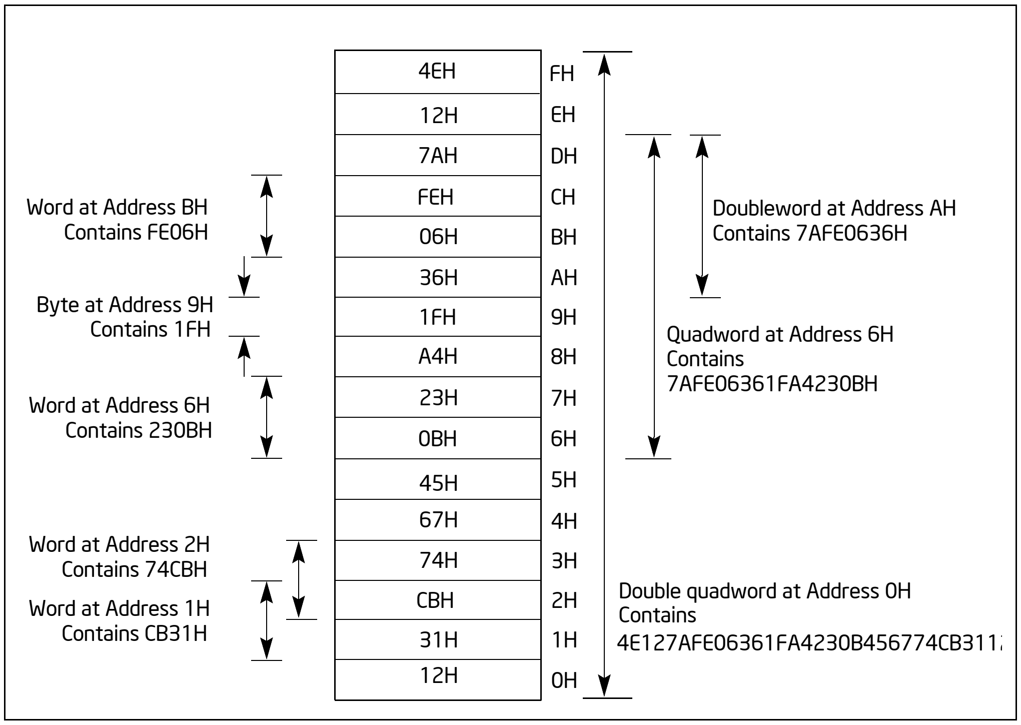

4EH
12H
7AH
FEH
06H
36H
1FH
A4H
23H
0BH
45H
67H
74H
CBH
31H
12H
FH
EH
DH
CH
BH
AH
9H
8H
7H
6H
5H
4H
3H
2H
1H
0H
Word at Address BH Contains FE06H
Byte at Address 9H Contains 1FH
Word at Address 6H Contains 230BH
Word at Address 2H Contains 74CBH
Word at Address 1H Contains CB31H
Doubleword at Address AH Contains 7AFE0636H
Quadword at Address 6H Contains 7AFE06361FA4230BH
Double quadword at Address 0H Contains 4E127AFE06361FA4230B456774CB311

Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, doublewords, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

Some instructions that operate on double quadwords require memory operands to be aligned on a natural boundary. These instructions generate a general-protection exception (#GP) if an unaligned operand is specified. A natural boundary for a double quadword is any address evenly divisible by 16. Other instructions that operate on double quadwords permit unaligned access (without generating a general-protection exception). However, additional memory bus cycles are required to access unaligned data from memory.

4.2 NUMERIC DATA TYPES

Although bytes, words, and doublewords are fundamental data types, some instructions support additional interpretations of these data types to allow operations to be performed on numeric data types (signed and unsigned integers, and floating-point numbers). Single precision (32-bit) floating-point and double precision (64-bit) floating-point data types are supported across all generations of Intel SSE extensions and Intel AVX extensions. The half precision (16-bit) floating-point data type was supported only with F16C extensions (VCVTPH2PS and VCVTPS2PH) beginning with the third generation of Intel® Core™ processors based on Ivy Bridge microarchitecture. Starting with the 4th generation Intel® Xeon® Scalable Processor Family, an Intel® AVX-512 instruction set architecture (ISA) for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16), which complements the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products. This ISA also provides complex-valued native hardware support for half precision floating-point. See Figure 4-3.

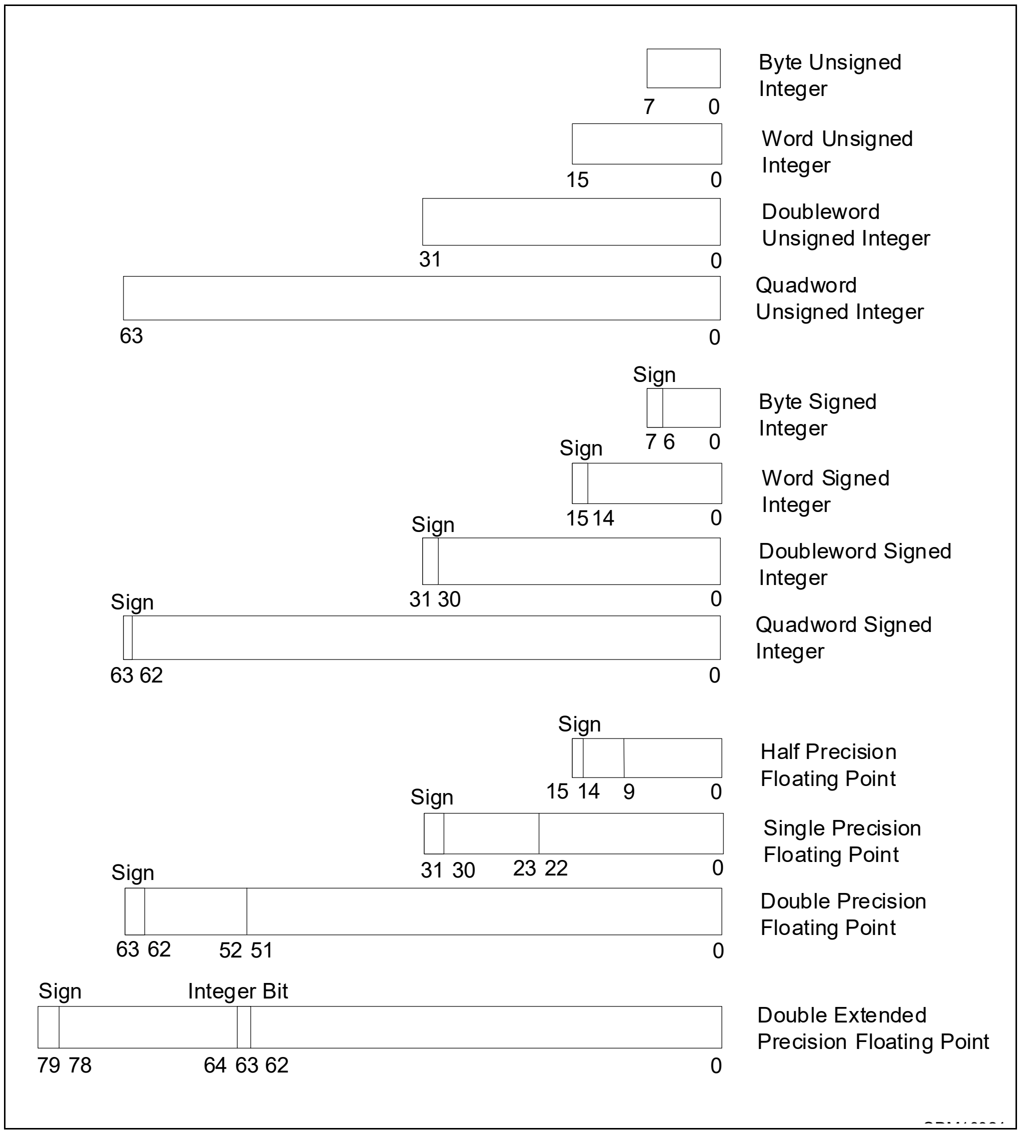


Figure 4-3. Numeric Data Types

4.2.1 Integers

The Intel 64 and IA-32 architectures define two types of integers: unsigned and signed. Unsigned integers are ordinary binary values ranging from 0 to the maximum positive number that can be encoded in the selected operand size. Signed integers are two's complement binary values that can be used to represent both positive and negative integer values.

Some integer instructions (such as the ADD, SUB, PADDB, and PSUBB instructions) operate on either unsigned or signed integer operands. Other integer instructions (such as IMUL, MUL, IDIV, DIV, FIADD, and FISUB) operate on only one integer type.

The following sections describe the encodings and ranges of the two types of integers.

4.2.1.1 Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, word, doubleword, and quadword. Their values range from 0 to 255 for an unsigned byte integer, from 0 to 65,535 for an unsigned word integer, from 0 to $2^{32} - 1$ for an unsigned doubleword integer, and from 0 to $2^{64} - 1$ for an unsigned quadword integer. Unsigned integers are sometimes referred to as **ordinals**.

4.2.1.2 Signed Integers

Signed integers are signed binary numbers held in a byte, word, doubleword, or quadword. All operations on signed integers assume a two's complement representation. The sign bit is located in bit 7 in a byte integer, bit 15 in a word integer, bit 31 in a doubleword integer, and bit 63 in a quadword integer (see the signed integer encodings in Table 4-1).

Table 4-1. Signed Integer Encodings

Class		Two's Complement Encoding	
		Sign	
Positive	Largest	0	11..11
		.	.
		.	.
	Smallest	0	00..01
Zero		0	00..00
Negative	Smallest	1	11..11
		.	.
		.	.
	Largest	1	00..00
Integer indefinite		1	00..00
		Signed Byte Integer:	← 7 bits →
		Signed Word Integer:	← 15 bits →
		Signed Doubleword Integer:	← 31 bits →
		Signed Quadword Integer:	← 63 bits →

The sign bit is set for negative integers and cleared for positive integers and zero. Integer values range from –128 to +127 for a byte integer, from –32,768 to +32,767 for a word integer, from -2^{31} to $+2^{31} - 1$ for a doubleword integer, and from -2^{63} to $+2^{63} - 1$ for a quadword integer.

When storing integer values in memory, word integers are stored in 2 consecutive bytes; doubleword integers are stored in 4 consecutive bytes; and quadword integers are stored in 8 consecutive bytes.

The integer indefinite is a special value that is sometimes returned by the x87 FPU when operating on integer values. For more information, see Section 8.2.1, "Indefinites."

4.2.2 Floating-Point Data Types

The IA-32 architecture defines and operates on four floating-point data types: half precision floating-point, single precision floating-point, double precision floating-point, and double-extended precision floating-point (see Figure 4-3). The data formats for these data types correspond directly to formats specified in the IEEE Standard 754 for Floating-Point Arithmetic.

The half precision (16-bit) floating-point data type was supported only with F16C extensions (VCVTPH2PS and VCVTPS2PH) beginning with the third generation of Intel Core processors based on Ivy Bridge microarchitecture. Starting with the 4th generation Intel Xeon Scalable Processor Family, an Intel AVX-512 instruction set architecture (ISA) for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in the IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16), which complements the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products.

Table 4-2 gives the length, precision, and approximate normalized range that can be represented by each of these data types. Denormal values are also supported in each of these types.

Table 4-2. Length, Precision, and Range of Floating-Point Data Types

Data Type	Length (Bits)	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Half Precision	16	11	2^{-14} to 2^{16}	6.10×10^{-5} to 6.55×10^{4}
Single Precision	32	24	2^{-126} to 2^{128}	1.18×10^{-38} to 3.40×10^{38}
Double Precision	64	53	2^{-1022} to 2^{1024}	2.23×10^{-308} to 1.80×10^{308}
Double-Extended Precision	80	64	2^{-16382} to 2^{16384}	3.36×10^{-4932} to 1.19×10^{4932}

NOTE

Section 4.8, “Real Numbers and Floating-Point Formats,” gives an overview of the IEEE Standard 754 floating-point formats and defines the terms integer bit, QNaN, SNaN, and denormal value.

Table 4-3 shows the floating-point encodings for zeros, denormalized finite numbers, normalized finite numbers, infinites, and NaNs for each of the three floating-point data types. It also gives the format for the QNaN floating-point indefinite value. (See Section 4.8.3.7, “QNaN Floating-Point Indefinite,” for a discussion of the use of the QNaN floating-point indefinite value.)

For the half precision, single precision, and double precision formats, only the fraction part of the significand is encoded. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. For the double extended precision format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinities, and NaNs, and to 0 for zero and denormalized numbers.

Table 4-3. Floating-Point Number and NaN Encodings

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
Positive	+∞	0	11..11	1	00..00
	+Normals	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+Denormals	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+Zero	0	00..00	0	00..00
Negative	−Zero	1	00..00	0	00..00
	−Denormals	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	−Normals	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	-∞	1	11..11	1	00..00

Table 4-3. Floating-Point Number and NaN Encodings (Contd.)

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	QNaN Floating-Point Indefinite	1	11..11	1	10..00
	Half Precision Single Precision: Double Precision: Double Extended Precision:		← 5 Bits → ← 8 Bits → ← 11 Bits → ← 15 Bits →		← 10 Bits → ← 23 Bits → ← 52 Bits → ← 63 Bits →

NOTES:

1. Integer bit is implied and not stored for half precision, single precision, and double precision formats.
2. The fraction for SNaN encodings must be non-zero with the most-significant bit 0.

The exponent of each floating-point data type is encoded in biased format; see Section 4.8.2.2, "Biased Exponent." The biasing constant is 15 for the half precision format, 127 for the single precision format, 1023 for the double precision format, and 16,383 for the double extended precision format.

When storing floating-point values in memory, half precision values are stored in 2 consecutive bytes in memory; single precision values are stored in 4 consecutive bytes in memory; double precision values are stored in 8 consecutive bytes; and double extended precision values are stored in 10 consecutive bytes.

The single precision and double precision floating-point data types are operated on by x87 FPU, and Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions. The double extended precision floating-point format is only operated on by the x87 FPU. See Section 11.6.8, "Compatibility of SIMD and x87 FPU Floating-Point Data Types," for a discussion of the compatibility of single precision and double precision floating-point data types between the x87 FPU and Intel SSE/SSE2/SSE3 extensions.

4.2.3 Brain Float16

Brain Float16 (BF16 or bfloat16) is a shortened 16-bit version of the IEEE Standard 754 floating-point 32-bit format. It aims to speed up training and inference for AI workloads. Figure 4-4 illustrates BF16 versus FP16 and FP32.

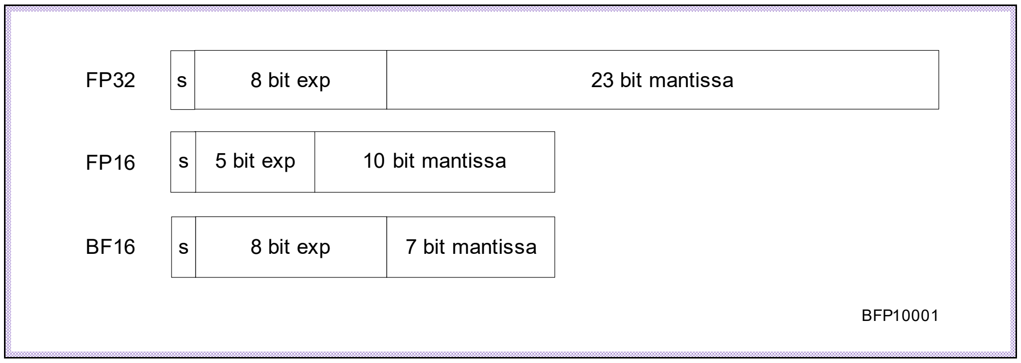


Figure 4-4. Comparison of BF16 to FP16 and FP32

4.2.3.1 Numeric Definition

BF16 has one sign bit, eight exponent bits, and seven mantissa bits.

Table 4-4. BF16 Format Numeric Definitions

Number	BF16 (E8M7)
Exponent Bias	127
Maximum Normal	S 11111110 11111111 = $\pm 2^{127} * 1.99 \sim= \pm 3.39 * 10^{38}$
Minimum Normal	S 00000001 00000000 = $\pm 2^{-126} \sim= \pm 1.18 * 10^{-38}$
Maximum Denormal	S 00000000 1111111 = $\pm 2^{-126} * 0.99 \sim= \pm 1.17 * 10^{-38}$
Minimum Denormal	S 00000000 0000001 = $\pm 2^{-133} \sim= \pm 9.18 * 10^{-41}$
NaNs	S 11111111 {non-zero}
Infinity	S 11111111 0000000
Zeros	S 00000000 0000000

Although denormal values are shown in the table, they are not used in computations (see Section 4.2.3.2).

4.2.3.2 Rounding, Denormal Handling, and FP Exceptions

Intel architecture supports BF16 in AMX dot product instructions, AVX dot product instructions, and AVX convert instructions.

- Rounding: All operations are executed using Round to nearest (even) mode (e.g., for convert instructions).
- Denormal Handling: For BF16, input denormal values are replaced with zeros and FP32 underflow results are flushed to zero. All instructions that accept BF16 inputs have FP32 results.
- Floating-point exceptions:
  - Instructions operating on BF16 values neither consult nor update the MXCSR.
  - Instructions operating on BF16 values do not raise exceptions.

4.3 POINTER DATA TYPES

Pointers are addresses of locations in memory.

In non-64-bit modes, the architecture defines two types of pointers: a **near pointer** and a **far pointer**. A near pointer is a 32-bit (or 16-bit) offset (also called an **effective address**) within a segment. Near pointers are used for all memory references in a flat memory model or for references in a segmented model where the identity of the segment being accessed is implied.

A far pointer is a logical address, consisting of a 16-bit segment selector and a 32-bit (or 16-bit) offset. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly. Near and far pointers with 32-bit offsets are shown in Figure 4-5.

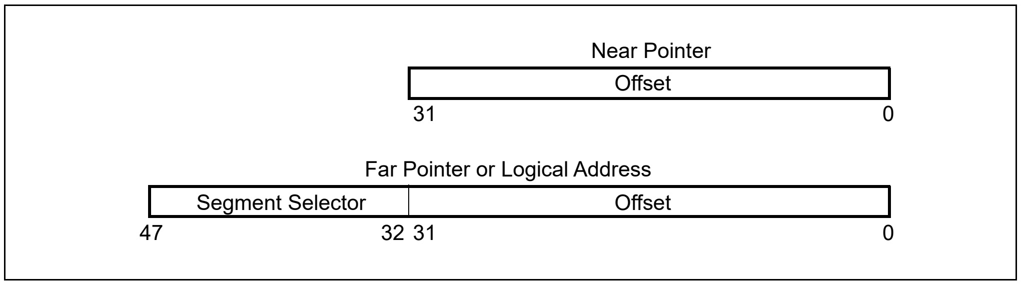


Figure 4-5. Pointer Data Types

4.3.1 Pointer Data Types in 64-Bit Mode

In 64-bit mode (a sub-mode of IA-32e mode), a **near pointer** is 64 bits. This equates to an effective address. **Far pointers** in 64-bit mode can be one of three forms:

- 16-bit segment selector, 16-bit offset if the operand size is 32 bits.
- 16-bit segment selector, 32-bit offset if the operand size is 32 bits.
- 16-bit segment selector, 64-bit offset if the operand size is 64 bits.

See Figure 4-6.

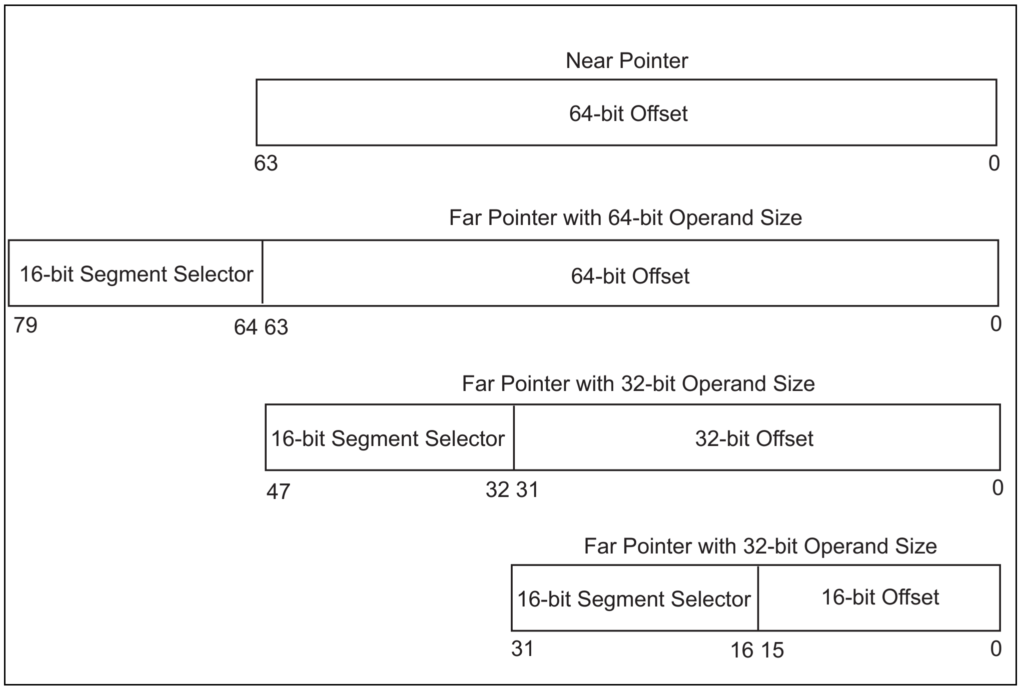


Figure 4-6. Pointers in 64-Bit Mode

4.4 BIT FIELD DATA TYPE

A **bit field** (see Figure 4-7) is a contiguous sequence of bits. It can begin at any bit position of any byte in memory and can contain up to 32 bits.

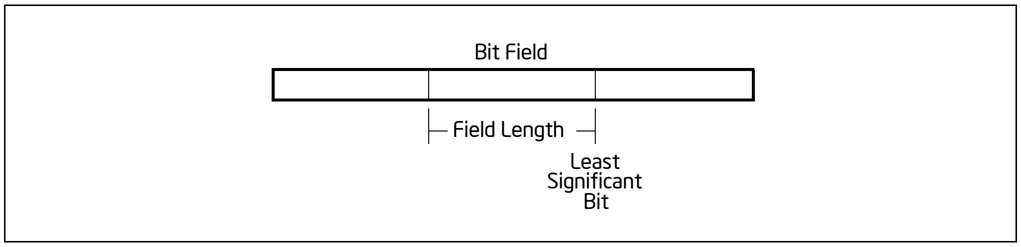


Figure 4-7. Bit Field Data Type

4.5 STRING DATA TYPES

Strings are continuous sequences of bits, bytes, words, or doublewords. A **bit string** can begin at any bit position of any byte and can contain up to $2^{32} – 1$ bits. A **byte string** can contain bytes, words, or doublewords and can range from zero to $2^{32} – 1$ bytes (4 GBytes).

4.6 PACKED SIMD DATA TYPES

Intel 64 and IA-32 architectures define and operate on a set of 64-bit and 128-bit packed data type for use in SIMD operations. These data types consist of fundamental data types (packed bytes, words, doublewords, and quadwords) and numeric interpretations of fundamental types for use in packed integer and packed floating-point operations.

4.6.1 64-Bit SIMD Packed Data Types

The 64-bit packed SIMD data types were introduced into the IA-32 architecture in the Intel MMX technology. They are operated on in MMX registers. The fundamental 64-bit packed data types are packed bytes, packed words, and packed doublewords (see Figure 4-8). When performing numeric SIMD operations on these data types, these data types are interpreted as containing byte, word, or doubleword integer values.

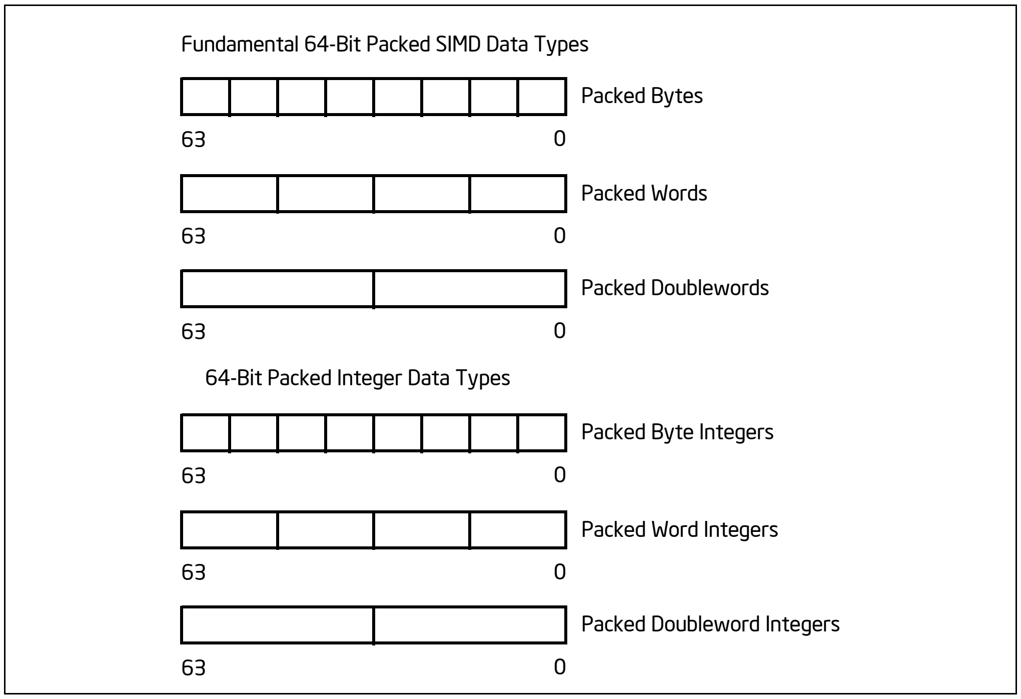


Figure 4-8. 64-Bit Packed SIMD Data Types

4.6.2 128-Bit Packed SIMD Data Types

The 128-bit packed SIMD data types were introduced into the IA-32 architecture in the Intel SSE extensions and used with Intel SSE2, SSE3, SSSE3, SSE4.1, and AVX extensions. They are operated on primarily in the 128-bit XMM registers and memory. The fundamental 128-bit packed data types are packed bytes, packed words, packed doublewords, and packed quadwords (see Figure 4-9). When performing SIMD operations on these fundamental data types in XMM registers, these data types are interpreted as containing packed or scalar half precision floating-point, single precision floating-point or double precision floating-point values, or as containing packed byte, word, doubleword, or quadword integer values.

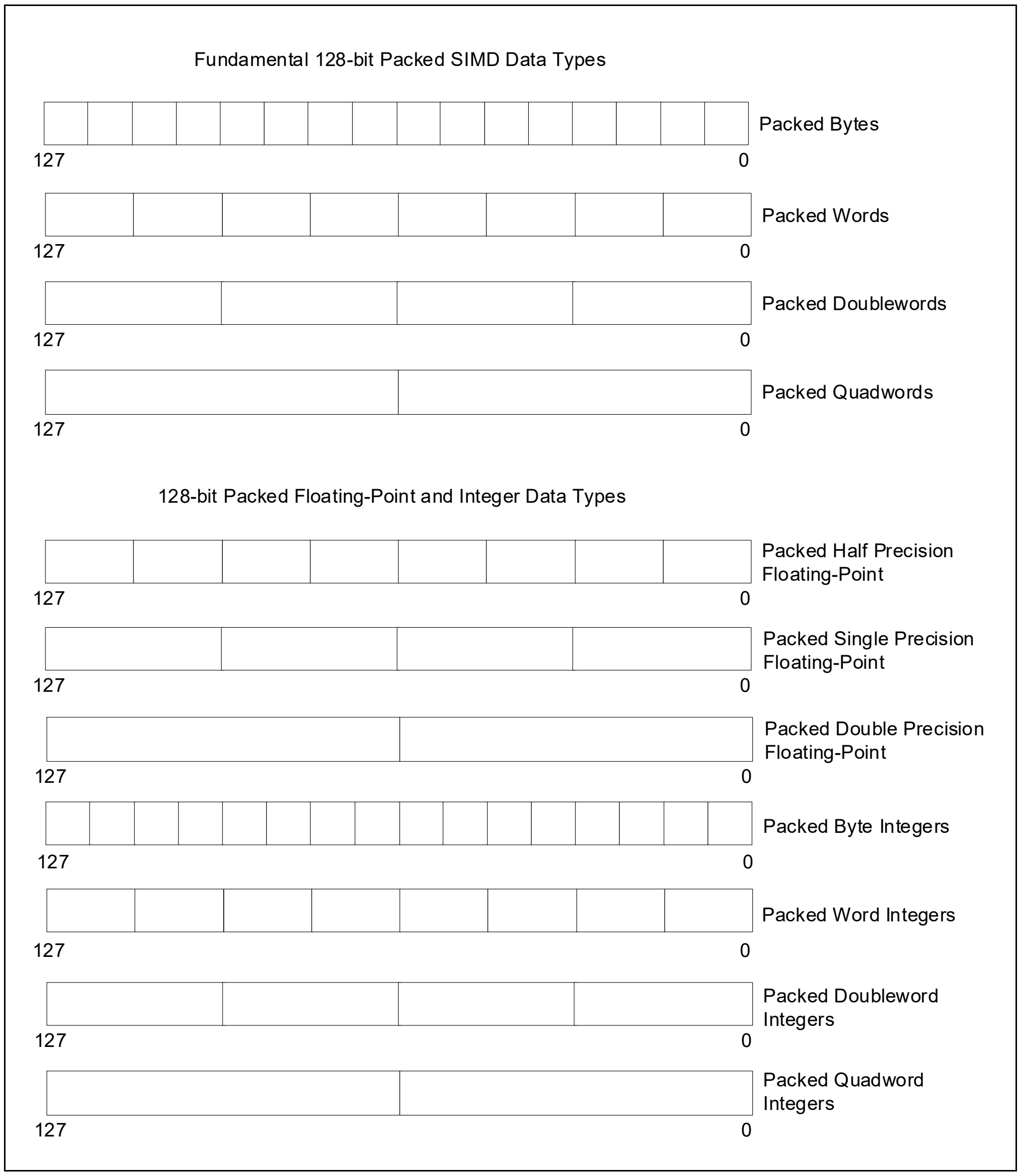

Fundamental 128-bit Packed SIMD Data Types
Packed Bytes
127
0
Packed Words
Packed Doublewords
Packed Quadwords
128-bit Packed Floating-Point and Integer Data Types
Packed Half Precision Floating-Point
Packed Single Precision Floating-Point
Packed Double Precision Floating-Point
Packed Byte Integers
Packed Word Integers
Packed Doubleword Integers
Packed Quadword Integers

Figure 4-9. 128-Bit Packed SIMD Data Types

4.7 BCD AND PACKED BCD INTEGERS

Binary-coded decimal integers (BCD integers) are unsigned 4-bit integers with valid values ranging from 0 to 9. IA-32 architecture defines operations on BCD integers located in one or more general-purpose registers or in one or more x87 FPU registers (see Figure 4-10).

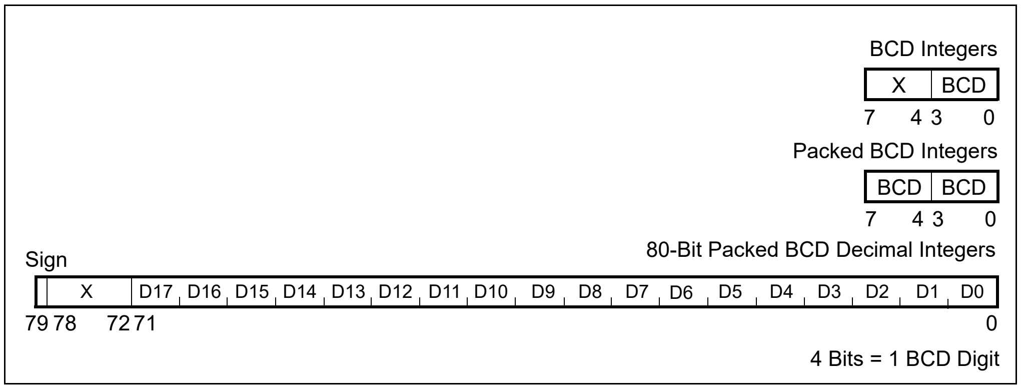

BCD Integers
X
BCD
7 4 3 0
Packed BCD Integers
BCD
BCD
7 4 3 0
80-Bit Packed BCD Decimal Integers
Sign
X
D17 D16 D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
79 78 72 71 0
4 Bits = 1 BCD Digit

Figure 4-10. BCD Data Types

When operating on BCD integers in general-purpose registers, the BCD values can be unpacked (one BCD digit per byte) or packed (two BCD digits per byte). The value of an unpacked BCD integer is the binary value of the low half-byte (bits 0 through 3). The high half-byte (bits 4 through 7) can be any value during addition and subtraction, but must be zero during multiplication and division. Packed BCD integers allow two BCD digits to be contained in one byte. Here, the digit in the high half-byte is more significant than the digit in the low half-byte.

When operating on BCD integers in x87 FPU data registers, BCD values are packed in an 80-bit format and referred to as decimal integers. In this format, the first 9 bytes hold 18 BCD digits, 2 digits per byte. The least-significant digit is contained in the lower half-byte of byte 0 and the most-significant digit is contained in the upper half-byte of byte 9. The most significant bit of byte 10 contains the sign bit (0 = positive and 1 = negative; bits 0 through 6 of byte 10 are don’t care bits). Negative decimal integers are not stored in two's complement form; they are distinguished from positive decimal integers only by the sign bit. The range of decimal integers that can be encoded in this format is $-10^{18} + 1$ to $10^{18} - 1$.

The decimal integer format exists in memory only. When a decimal integer is loaded in an x87 FPU data register, it is automatically converted to the double extended precision floating-point format. All decimal integers are exactly representable in double extended precision format.

Table 4-5 gives the possible encodings of value in the decimal integer data type.

Table 4-5. Packed Decimal Integer Encodings

Class	Sign		Magnitude					
			digit	digit	digit	digit	...	digit
Positive								
Largest	0	0000000	1001	1001	1001	1001	...	1001
	.	.			.			
	.	.			.			
Smallest	0	0000000	0000	0000	0000	0000	...	0001
Zero	0	0000000	0000	0000	0000	0000	...	0000
Negative								
Zero	1	0000000	0000	0000	0000	0000	...	0000
Smallest	1	0000000	0000	0000	0000	0000	...	0001
	.	.			.			
	.	.			.			
Largest	1	0000000	1001	1001	1001	1001	...	1001

Table 4-5. Packed Decimal Integer Encodings (Contd.)

Class	Sign		Magnitude					
			digit	digit	digit	digit	...	digit
Packed BCD Integer Indefinite	1	1111111	1111	1111	1100	0000	...	0000
	← 1 byte →		← 9 bytes →					

The packed BCD integer indefinite encoding (FFFFC000000000000000H) is stored by the FBSTP instruction in response to a masked floating-point invalid-operation exception. Attempting to load this value with the FBLD instruction produces an undefined result.

4.8 REAL NUMBERS AND FLOATING-POINT FORMATS

This section describes how real numbers are represented in floating-point format in x87 FPU and SSE/SSE2/SSE3/SSE4.1 and Intel AVX floating-point instructions. It also introduces terms such as normalized numbers, denormalized numbers, biased exponents, signed zeros, and NaNs. Readers who are already familiar with floating-point processing techniques and the IEEE Standard 754 for Floating-Point Arithmetic may wish to skip this section.

4.8.1 Real Number System

As shown in Figure 4-11, the real-number system comprises the continuum of real numbers from minus infinity (− ∞) to plus infinity (+ ∞).

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number (floating-point) calculations. As shown at the bottom of Figure 4-11, the subset of real numbers that the IA-32 architecture supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the IEEE Standard 754 floating-point formats.

4.8.2 Floating-Point Format

To increase the speed and efficiency of real-number computations, computers and microprocessors typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent (see Figure 4-12).

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a 1-bit binary integer (also referred to as the J-bit) and a binary fraction. The integer-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power by which the significand is multiplied.

Table 4-6 shows how the real number 178.125 (in ordinary decimal format) is stored in IEEE Standard 754 floating-point format. The table lists a progression of real number notations that leads to the single precision, 32-bit floating-point format. In this format, the significand is normalized (see Section 4.8.2.1, “Normalized Numbers”) and the exponent is biased (see Section 4.8.2.2, “Biased Exponent”). For the single precision floating-point format, the biasing constant is +127.

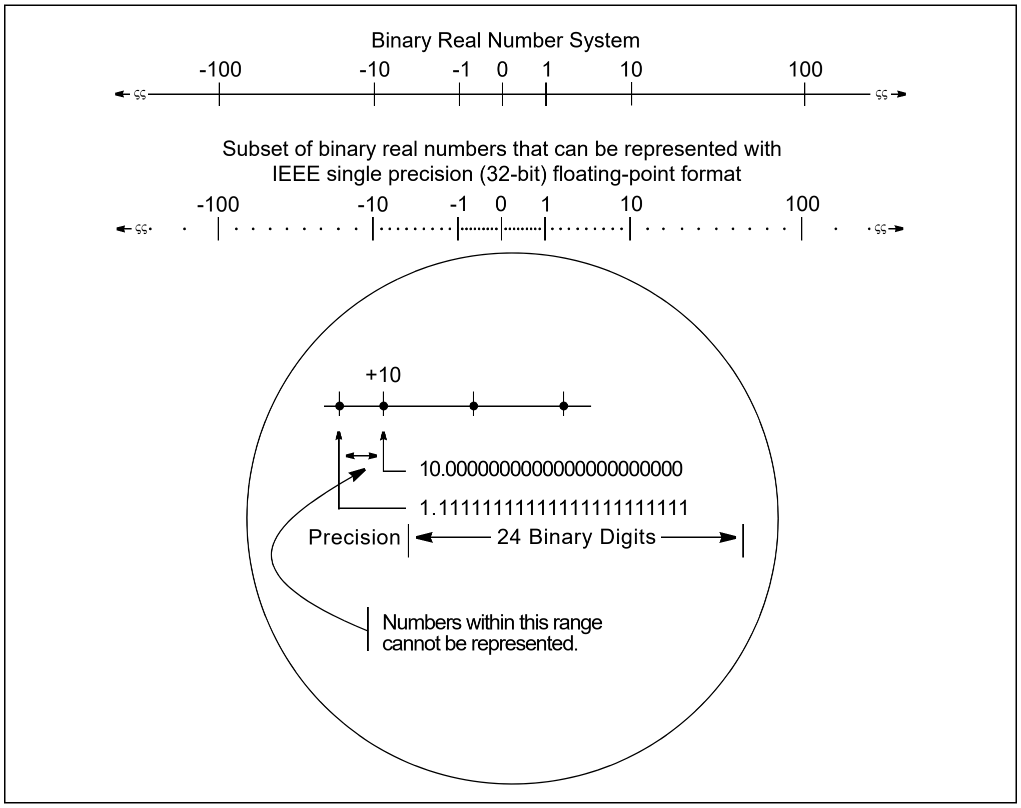

Binary Real Number System
-100
-10
-1
0
1
10
100
Subset of binary real numbers that can be represented with IEEE single precision (32-bit) floating-point format
+10
10.0000000000000000000000
1.11111111111111111111111
Precision
24 Binary Digits
Numbers within this range cannot be represented.

Figure 4-11. Binary Real Number System

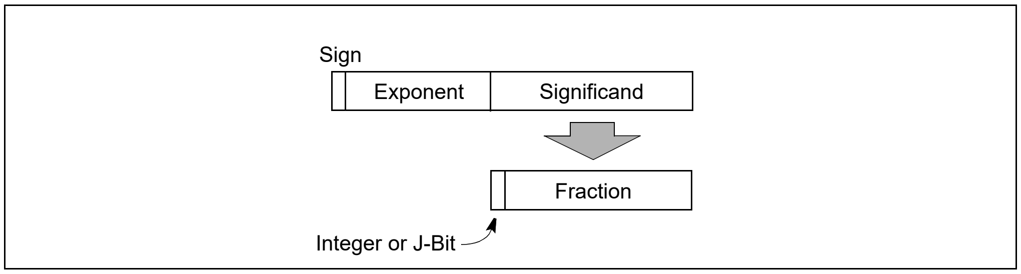


Figure 4-12. Binary Floating-Point Format

Table 4-6. Real and Floating-Point Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10} 2$		
Scientific Binary	$1.0110010001E_2 111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_2 10000110$		
IEEE Single Precision Format	Sign	Biased Exponent	Normalized Significand
	0	10000110	01100100010000000000000 1. (Implied)

4.8.2.1 Normalized Numbers

In most cases, floating-point numbers are encoded in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and the following fraction:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that specifies the number's binary point.

4.8.2.2 Biased Exponent

In the IA-32 architecture, the exponents of floating-point numbers are encoded in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

See Section 4.2.2, "Floating-Point Data Types," for a list of the biasing constants that the IA-32 architecture uses for the various sizes of floating-point data-types.

4.8.3 Real Number and Non-number Encodings

A variety of real numbers and special values can be encoded in the IEEE Standard 754 floating-point format. These numbers and values are generally divided into the following classes:

- Signed zeros
- Denormalized finite numbers
- Normalized finite numbers
- Signed infinities
- NaNs
- Indefinite numbers

(The term NaN stands for "Not a Number.")

Figure 4-13 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single precision floating-point format. The term "S" indicates the sign bit, "E" the biased exponent, and "Sig" the significand. The exponent values are given in decimal. The integer bit is shown for the significands, even though the integer bit is implied in single precision floating-point format.

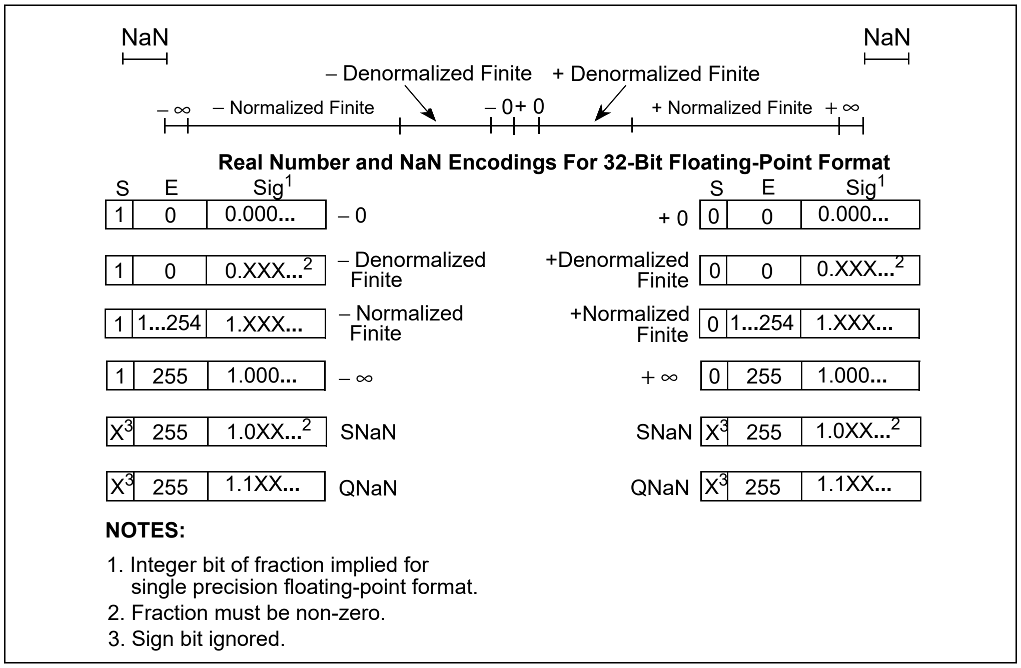

NaN
NaN
– Denormalized Finite
+ Denormalized Finite
– ∞
– Normalized Finite
– 0
+ 0
+ Normalized Finite
+ ∞
Real Number and NaN Encodings For 32-Bit Floating-Point Format
S E Sig¹
1 0 0.000... – 0
+ 0 0 0 0.000...
1 0 0.XXX...² – Denormalized Finite
+Denormalized Finite 0 0 0.XXX...²
1 1...254 1.XXX... – Normalized Finite
+Normalized Finite 0 1...254 1.XXX...
1 255 1.000... – ∞
+ ∞ 0 255 1.000...
X³ 255 1.0XX...² SNaN
SNaN X³ 255 1.0XX...²
X³ 255 1.1XX... QNaN
QNaN X³ 255 1.1XX...
NOTES:
1. Integer bit of fraction implied for single precision floating-point format.
2. Fraction must be non-zero.
3. Sign bit ignored.

Figure 4-13. Real Numbers and NaNs

An IA-32 processor can operate on and/or return any of these values, depending on the type of computation being performed. The following sections describe these number and non-number classes.

4.8.3.1 Signed Zeros

Zero can be represented as a +0 or a –0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

4.8.3.2 Normalized and Denormalized Finite Numbers

Non-zero, finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non-zero finite values that can be encoded in a normalized real number format between zero and ∞. In the single precision floating-point format shown in Figure 4-13, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased, the exponent range is from $–126_{10}$ to $+127_{10}$).

When floating-point numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called **denormalized** numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization may cause a loss of precision (the number of significant bits is reduced by the leading zeros).

When performing normalized floating-point computations, an IA-32 processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an **underflow** condition. The exact conditions are specified in Section 4.9.1.5, “Numeric Underflow Exception (#U).”

A denormalized number is computed through a technique called gradual underflow. Table 4-7 gives an example of gradual underflow in the denormalization process. Here the single precision format is being used, so the minimum exponent (unbiased) is $–126_{10}$. The true result in this example requires an exponent of $–129_{10}$ in order to have a normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 4-7. Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	–129	1.01011100000...00
Denormalize	0	–128	0.10101110000...00
Denormalize	0	–127	0.01010111000...00
Denormalize	0	–126	0.00101011100...00
Denormal Result	0	–126	0.00101011100...00

* Expressed as an unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

The Intel 64 and IA-32 architectures deal with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

4.8.3.3 Signed Infinities

The two infinities, $+\infty$ and $-\infty$, represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a significand of 1.00...00 (the integer bit may be implied) and the maximum biased exponent allowed in the specified format (for example, 255_{10} for the single precision format).

The signs of infinities are observed, and comparisons are possible. Infinities are always interpreted in the affine sense; that is, $-\infty$ is less than any finite number and $+\infty$ is greater than any finite number. Arithmetic on infinities is always exact. Exceptions are generated only when the use of an infinity as a source operand constitutes an invalid operation.

Whereas denormalized numbers may represent an underflow condition, the two ∞ numbers may represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

4.8.3.4 NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 4-13, the encoding space for NaNs in the floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction (the sign bit is ignored for NaNs).

The IA-32 architecture defines two classes of NaNs: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal a floating-point invalid-operation exception whenever they appear as operands in arithmetic operations.

SNaNs are typically used to trap or invoke an exception handler. They must be inserted by software; that is, the processor never generates an SNaN as a result of a floating-point operation.

4.8.3.5 Operating on SNaNs and QNaNs

When a floating-point operation is performed on an SNaN and/or a QNaN, the result of the operation is either a QNaN delivered to the destination operand or the generation of a floating-point invalid operation exception, depending on the following rules:

- If one of the source operands is an SNaN and the floating-point invalid-operation exception is not masked (see Section 4.9.1.1, "Invalid Operation Exception (#I)"), then a floating-point invalid-operation exception is signaled and no result is stored in the destination operand. If one of the source operands is a QNaN and the floating-point invalid-operation exception is not masked and the operation is one that generates an invalid-operation exception for QNaN operands as described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," or Section 11.5.2.1, "Invalid Operation Exception (#I)," then a floating-point invalid-operation exception is signaled and no result is stored in the destination operand.
- If either or both of the source operands are NaNs and floating-point invalid-operation exception is masked, the result is as shown in Table 4-8. When an SNaN is converted to a QNaN, the conversion is handled by setting the most-significant fraction bit of the SNaN to 1. Also, when one of the source operands is an SNaN, or when it is a QNaN and the operation is one that generates an invalid-operation exception for QNaN operands as described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," or Section 11.5.2.1, "Invalid Operation Exception (#I)," then the floating-point invalid-operation exception flag is set. Note that for some combinations of source operands, the result is different for x87 FPU operations and for Intel SSE/SSE2/SSE3/SSE4.1 operations. Intel AVX follows the same behavior as Intel SSE/SSE2/SSE3/SSE4.1 in this respect.
- When neither of the source operands is a NaN, but the operation generates a floating-point invalid-operation exception (see Tables 8-10 and 11-1), the result is commonly a QNaN FP Indefinite (Section 4.8.3.7).

Any exceptions to the behavior described in Table 4-8 are described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," and Section 11.5.2.1, "Invalid Operation Exception (#I)."

Table 4-8. Rules for Handling NaNs

Source Operands	Result[1]
SNaN and QNaN	X87 FPU — QNaN source operand. SSE/SSE2/SSE3/SSE4.1/AVX — First source operand (if this operand is an SNaN, it is converted to a QNaN).
Two SNaNs	X87 FPU — SNaN source operand with the larger significand, converted into a QNaN. SSE/SSE2/SSE3/SSE4.1/AVX — First source operand converted to a QNaN.
Two QNaNs	X87 FPU — QNaN source operand with the larger significand. SSE/SSE2/SSE3/SSE4.1/AVX — First source operand.
SNaN and a floating-point value	SNaN source operand, converted into a QNaN.
QNaN and a floating-point value	QNaN source operand.
SNaN (for instructions that take only one operand)	SNaN source operand, converted into a QNaN.
QNaN (for instructions that take only one operand)	QNaN source operand.

NOTE:

1. For SSE/SSE2/SSE3/SSE4.1 instructions, the first operand is generally a source operand that becomes the destination operand. For AVX instructions, the first source operand is usually the 2nd operand in a non-destructive source syntax. Within the **Result** column, the x87 FPU notation also applies to the FISTTP instruction in SSE3; the SSE3 notation applies to the SIMD floating-point instructions.

4.8.3.6 Using SNaNs and QNaNs in Applications

Except for the rules given at the beginning of Section 4.8.3.4, "NaNs," for encoding SNaNs and QNaNs, software is free to use the bits in the significand of a NaN for any purpose. Both SNaNs and QNaNs can be encoded to carry and store data, such as diagnostic information.

By unmasking the invalid operation exception, the programmer can use signaling NaNs to trap to the exception handler. The generality of this approach and the large number of NaN values that are available provide the sophisticated programmer with a tool that can be applied to a variety of special situations.

For example, a compiler can use signaling NaNs as references to uninitialized (real) array elements. The compiler can preinitialize each array element with a signaling NaN whose significand contains the index (relative position) of the element. Then, if an application program attempts to access an element that it has not initialized, it can use the NaN placed there by the compiler. If the invalid operation exception is unmasked, an interrupt will occur, and the exception handler will be invoked. The exception handler can determine which element has been accessed, since the operand address field of the exception pointer will point to the NaN, and the NaN will contain the index number of the array element.

Quiet NaNs are often used to speed up debugging. In its early testing phase, a program often contains multiple errors. An exception handler can be written to save diagnostic information in memory whenever it is invoked. After storing the diagnostic data, it can supply a quiet NaN as the result of the erroneous instruction, and that NaN can point to its associated diagnostic area in memory. The program will then continue, creating a different NaN for each error. When the program ends, the NaN results can be used to access the diagnostic data saved at the time the errors occurred. Many errors can thus be diagnosed and corrected in one test run.

In embedded applications that use computed results in further computations, an undetected QNaN can invalidate all subsequent results. Such applications should therefore periodically check for QNaNs and provide a recovery mechanism to be used if a QNaN result is detected.

4.8.3.7 QNaN Floating-Point Indefinite

For the floating-point data type encodings (single precision, double precision, and double extended precision), one unique encoding (a QNaN) is reserved for representing the special value QNaN floating-point indefinite. The x87 FPU and the Intel SSE/SSE2/SSE3/SSE4.1/AVX extensions return these indefinite values as responses to some masked floating-point exceptions. Table 4-3 shows the encoding used for the QNaN floating-point indefinite.

4.8.3.8 Half Precision Floating-Point Operation

Two instructions, VCVTPH2PS and VCVTPS2PH, which provide conversion only between half precision and single precision floating-point values, were introduced with the F16C extensions beginning with the third generation of Intel Core processors based on Ivy Bridge microarchitecture. Starting with the 4th generation Intel Xeon Scalable Processor Family, an Intel AVX-512 instruction set architecture (ISA) for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in the IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16). These additions complement the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products.

The SIMD floating-point exception behavior of the VCVTPH2PS and VCVTPS2PH instructions, as well as of the other half precision instructions, are described in Section 14.4.1.

4.8.4 Rounding

When performing floating-point operations, the processor produces an infinitely precise floating-point result in the destination format (half precision, single precision, double precision, or double extended precision floating-point) whenever possible. However, because only a subset of the numbers in the real number continuum can be represented in IEEE Standard 754 floating-point formats, it is often the case that an infinitely precise result cannot be encoded exactly in the format of the destination operand.

For example, the following value (*a*) has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the single precision format (which has only a 23-bit fraction):

(*a*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011\underline{1}E_2\ 101$

To round this result (*a*), the processor first selects two representable fractions *b* and *c* that most closely bracket *a* in value ($b < a < c$).

(*b*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011E_2\ 101$

(*c*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 100E_2\ 101$

The processor then sets the result to *b* or to *c* according to the selected rounding mode. Rounding introduces an error in a result that is less than one unit in the last place (the least significant bit position of the floating-point value) to which the result is rounded.

The IEEE Standard 754 defines four rounding modes (see Table 4-9): round to nearest, round up, round down, and round toward zero. The default rounding mode (for the Intel 64 and IA-32 architectures) is round to nearest. This mode provides the most accurate and statistically unbiased estimate of the true result and is suitable for most applications.

Table 4-9. Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero). Default
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

The round up and round down modes are termed **directed rounding** and can be used to implement interval arithmetic. Interval arithmetic is used to determine upper and lower bounds for the true result of a multistep computation, when the intermediate results of the computation are subject to rounding.

The round toward zero mode (sometimes called the "chop" mode) is commonly used when performing integer arithmetic with the x87 FPU.

The rounded result is called the inexact result. When the processor produces an inexact result, the floating-point precision (inexact) flag (PE) is set (see Section 4.9.1.6, "Inexact-Result (Precision) Exception (#P)").

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

4.8.4.1 Rounding Control (RC) Fields

In the Intel 64 and IA-32 architectures, the rounding mode is controlled by a 2-bit rounding-control (RC) field (Table 4-9 shows the encoding of this field). The RC field is implemented in two different locations:

- X87 FPU control register (bits 10 and 11).
- The MXCSR register (bits 13 and 14).

Although these two RC fields perform the same function, they control rounding for different execution environments within the processor. The RC field in the x87 FPU control register controls rounding for computations performed with the x87 FPU instructions; the RC field in the MXCSR register controls rounding for SIMD floating-point computations performed with the Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions.

4.8.4.2 Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions

The following Intel SSE/SSE2 instructions automatically truncate the results of conversions from floating-point values to integers when the result it inexact: CVTTPD2DQ, CVTTPS2DQ, CVTTPD2PI, CVTTPS2PI, CVTTSD2SI, and CVTTSS2SI. Here, truncation means the round toward zero mode described in Table 4-9. There are also several Intel AVX2 and AVX-512 instructions which use truncation (VCVTT*).

4.9 OVERVIEW OF FLOATING-POINT EXCEPTIONS

The following section provides an overview of floating-point exceptions and their handling in the IA-32 architecture. For information specific to the x87 FPU and to the Intel SSE/SSE2/SSE3/SSE4.1/AVX extensions, refer to the following sections:

- Section 4.9, "Overview of Floating-Point Exceptions."
- Section 11.5, "Intel® SSE, SSE2, and SSE3 Exceptions."
- Section 12.8.4, "IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions."
- Section 14.10, "SIMD Floating-Point Exceptions."

When operating on floating-point operands, the IA-32 architecture recognizes and detects six classes of exception conditions:

- Invalid operation (#I).
- Divide-by-zero (#Z).
- Denormalized operand (#D).
- Numeric overflow (#O).
- Numeric underflow (#U).
- Inexact result (precision) (#P).

The nomenclature of "#" symbol followed by one or two letters (for example, #P) is used in this manual to indicate exception conditions. It is merely a short-hand form and is not related to assembler mnemonics.

NOTE

All of the exceptions listed above except the denormal-operand exception (#D) are defined in IEEE Standard 754.

The invalid-operation, divide-by-zero and denormal-operand exceptions are pre-computation exceptions (that is, they are detected before any arithmetic operation occurs). The numeric-underflow, numeric-overflow and precision exceptions are post-computation exceptions.

Each of the six exception classes has a corresponding flag bit (IE, ZE, OE, UE, DE, or PE) and mask bit (IM, ZM, OM, UM, DM, or PM). When one or more floating-point exception conditions are detected, the processor sets the appropriate flag bits, then takes one of two possible courses of action, depending on the settings of the corresponding mask bits:

- Mask bit set. Handles the exception automatically, producing a predefined (and often times usable) result, while allowing program execution to continue undisturbed.
- Mask bit clear. Invokes a software exception handler to handle the exception.

The masked (default) responses to exceptions have been chosen to deliver a reasonable result for each exception condition and are generally satisfactory for most floating-point applications. By masking or unmasking specific floating-point exceptions, programmers can delegate responsibility for most exceptions to the processor and reserve the most severe exception conditions for software exception handlers.

Because the exception flags are "sticky," they provide a cumulative record of the exceptions that have occurred since they were last cleared. A programmer can thus mask all exceptions, run a calculation, and then inspect the exception flags to see if any exceptions were detected during the calculation.

In the IA-32 architecture, floating-point exception flag and mask bits are implemented in two different locations:

- X87 FPU status word and control word. The flag bits are located at bits 0 through 5 of the x87 FPU status word and the mask bits are located at bits 0 through 5 of the x87 FPU control word (see Figures 8-4 and 8-6).
- MXCSR register. The flag bits are located at bits 0 through 5 of the MXCSR register and the mask bits are located at bits 7 through 12 of the register (see Figure 10-3).

Although these two sets of flag and mask bits perform the same function, they report on and control exceptions for different execution environments within the processor. The flag and mask bits in the x87 FPU status and control words control exception reporting and masking for computations performed with the x87 FPU instructions; the companion bits in the MXCSR register control exception reporting and masking for SIMD floating-point computations performed with the Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions.

Note that when exceptions are masked, the processor may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the processor can detect a denormalized operand, perform its masked response to this exception, and then detect numeric underflow.

See Section 4.9.2, "Floating-Point Exception Priority," for a description of the rules for exception precedence when more than one floating-point exception condition is detected for an instruction.

4.9.1 Floating-Point Exception Conditions

The following sections describe the various conditions that cause a floating-point exception to be generated and the masked response of the processor when these conditions are detected. The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C, & 3D, lists the floating-point exceptions that can be signaled for each floating-point instruction.

4.9.1.1 Invalid Operation Exception (#I)

The processor reports an invalid operation exception in response to one or more invalid arithmetic operands. If the invalid operation exception is masked, the processor sets the IE flag and returns an indefinite value or a QNaN. This value overwrites the destination register specified by the instruction. If the invalid operation exception is not masked, the IE flag is set, a software exception handler is invoked, and the operands remain unaltered.

See Section 4.8.3.6, "Using SNaNs and QNaNs in Applications," for information about the result returned when an exception is caused by an SNaN.

The processor can detect a variety of invalid arithmetic operations that can be coded in a program. These operations generally indicate a programming error, such as dividing ∞ by ∞. See the following sections for information regarding the invalid-operation exception when detected while executing x87 FPU or Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:

- X87 FPU; Section 8.5.1, "Invalid Operation Exception."
- SIMD floating-point exceptions; Section 11.5.2.1, "Invalid Operation Exception (#I)."
- Section 12.8.4, "IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions."
- Section 14.10, "SIMD Floating-Point Exceptions."

4.9.1.2 Denormal Operand Exception (#D)

The processor reports the denormal-operand exception if an arithmetic instruction attempts to operate on a denormal operand (see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers"). When the exception is masked, the processor sets the DE flag and proceeds with the instruction. Operating on denormal numbers will produce results at least as good as, and often better than, what can be obtained when denormal numbers are flushed to zero. Programmers can mask this exception so that a computation may proceed, then analyze any loss of accuracy when the final result is delivered.

When a denormal-operand exception is not masked, the DE flag is set, a software exception handler is invoked, and the operands remain unaltered. When denormal operands have reduced significance due to loss of low-order bits, it may be advisable to not operate on them. Precluding denormal operands from computations can be accomplished by an exception handler that responds to unmasked denormal-operand exceptions.

See the following sections for information regarding the denormal-operand exception when detected while executing x87 FPU or Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:

- X87 FPU; Section 8.5.2, "Denormal Operand Exception (#D)."
- SIMD floating-point exceptions; Section 11.5.2.2, "Denormal-Operand Exception (#D)."
- Section 12.8.4, "IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions."
- Section 14.10, "SIMD Floating-Point Exceptions."

4.9.1.3 Divide-By-Zero Exception (#Z)

The processor reports the floating-point divide-by-zero exception whenever an instruction attempts to divide a finite non-zero operand by 0. The masked response for the divide-by-zero exception is to set the ZE flag and return an infinity signed with the exclusive OR of the sign of the operands. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked, and the operands remain unaltered.

See the following sections for information regarding the divide-by-zero exception when detected while executing x87 FPU or Intel SSE/SSE2/AVX instructions:

- X87 FPU; Section 8.5.3, "Divide-By-Zero Exception (#Z)."
- SIMD floating-point exceptions; Section 11.5.2.3, "Divide-By-Zero Exception (#Z)."
- Section 12.8.4, "IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions."
- Section 14.10, "SIMD Floating-Point Exceptions."

4.9.1.4 Numeric Overflow Exception (#O)

The processor reports a floating-point numeric overflow exception whenever the rounded result of an instruction exceeds the largest allowable finite value that will fit into the destination operand. Table 4-10 shows the threshold range for numeric overflow for each of the floating-point formats; overflow occurs when a rounded result falls at or outside this threshold range.

Table 4-10. Numeric Overflow Thresholds

Floating-Point Format	Overflow Thresholds
Half Precision	$\lvert x \rvert \geq 1.0 * 2^{16}$
Single Precision	$\lvert x \rvert \geq 1.0 * 2^{128}$
Double Precision	$\lvert x \rvert \geq 1.0 * 2^{1024}$
Double Extended Precision	$\lvert x \rvert \geq 1.0 * 2^{16384}$

When a numeric-overflow exception occurs and the exception is masked, the processor sets the OE flag and returns one of the values shown in Table 4-11, according to the current rounding mode. See Section 4.8.4, "Rounding."

When numeric overflow occurs and the numeric-overflow exception is not masked, the OE flag is set, a software exception handler is invoked, and the source and destination operands either remain unchanged or a biased result is stored in the destination operand (depending whether the overflow exception was generated during an Intel SSE/SSE2/SSE3/SSE4.1/AVX floating-point operation or an x87 FPU operation).

Table 4-11. Masked Responses to Numeric Overflow

Rounding Mode	Sign of True Result	Result
To nearest	+	+∞
	–	–∞
Toward –∞	+	Largest finite positive number
	–	–∞
Toward +∞	+	+∞
	–	Largest finite negative number
Toward zero	+	Largest finite positive number
	–	Largest finite negative number

See the following sections for information regarding the numeric overflow exception when detected while executing x87 FPU instructions or while executing Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:

- X87 FPU; Section 8.5.4, "Numeric Overflow Exception (#O)."
- SIMD floating-point exceptions; Section 11.5.2.4, "Numeric Overflow Exception (#O)."

- Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
- Section 14.10, “SIMD Floating-Point Exceptions.”

4.9.1.5 Numeric Underflow Exception (#U)

The processor detects a potential floating-point numeric underflow condition whenever the result of rounding with unbounded exponent (taking into account precision control for x87) is non-zero and tiny; that is, non-zero and less than the smallest possible normalized, finite value that will fit into the destination operand. Table 4-12 shows the threshold range for numeric underflow for each of the floating-point formats (assuming normalized results); underflow occurs when a rounded result falls strictly within the threshold range. The ability to detect and handle underflow is provided to prevent a very small result from propagating through a computation and causing another exception (such as overflow during division) to be generated at a later time. Results which trigger underflow are also potentially less accurate.

Table 4-12. Numeric Underflow (Normalized) Thresholds

Floating-Point Format	Underflow Thresholds[1]
Half Precision	$\|x\| < 1.0 * 2^{-14}$
Single Precision	$\|x\| < 1.0 * 2^{-126}$
Double Precision	$\|x\| < 1.0 * 2^{-1022}$
Double Extended Precision	$\|x\| < 1.0 * 2^{-16382}$

NOTES:
1. Where ‘x’ is the result rounded to destination precision with an unbounded exponent range.

How the processor handles an underflow condition, depends on two related conditions:

- Creation of a tiny, non-zero result.
- Creation of an inexact result; that is, a result that cannot be represented exactly in the destination format.

Which of these events causes an underflow exception to be reported and how the processor responds to the exception condition depends on whether the underflow exception is masked:

- **Underflow exception masked —** The underflow exception is reported (the UE flag is set) only when the result is both tiny and inexact. The processor returns a correctly signed result whose magnitude is less than or equal to the smallest positive normal floating-point number to the destination operand, regardless of inexactness.
- **Underflow exception not masked —** The underflow exception is reported when the result is non-zero tiny, regardless of inexactness. The processor leaves the source and destination operands unaltered or stores a biased result in the destination operand (depending whether the underflow exception was generated during an Intel SSE/SSE2/SSE3/AVX floating-point operation or an x87 FPU operation) and invokes a software exception handler.

See the following sections for information regarding the numeric underflow exception when detected while executing x87 FPU instructions or while executing Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:

- X87 FPU; Section 8.5.5, “Numeric Underflow Exception (#U).”
- SIMD floating-point exceptions; Section 11.5.2.5, “Numeric Underflow Exception (#U).”
- Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
- Section 14.10, “SIMD Floating-Point Exceptions.”

4.9.1.6 Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary floating-point form. This exception occurs frequently and indicates that some (normally acceptable) accuracy will be lost due to rounding. The exception is supported for applications that need to perform exact arithmetic only. Because the rounded result is generally satisfactory for most applications, this exception is commonly masked.

If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or underflow condition has not occurred, the processor sets the PE flag and stores the rounded result in the destination operand. The current rounding mode determines the method used to round the result. See Section 4.8.4, “Rounding.”

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has not occurred, the PE flag is set, the rounded result is stored in the destination operand, and a software exception handler is invoked.

If an inexact result occurs in conjunction with numeric overflow or underflow, one of the following operations is carried out:

- If an inexact result occurs along with masked overflow or underflow, the OE flag or UE flag and the PE flag are set and the result is stored as described for the overflow or underflow exceptions; see Section 4.9.1.4, “Numeric Overflow Exception (#O),” or Section 4.9.1.5, “Numeric Underflow Exception (#U).” If the inexact result exception is unmasked, the processor also invokes a software exception handler.
- If an inexact result occurs along with unmasked overflow or underflow and the destination operand is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or underflow exceptions, and a software exception handler is invoked.

If an unmasked numeric overflow or underflow exception occurs and the destination operand is a memory location (which can happen only for a floating-point store), the inexact-result condition is not reported and the C1 flag is cleared.

See the following sections for information regarding the inexact-result exception when detected while executing x87 FPU or Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:

- X87 FPU; Section 8.5.6, “Inexact-Result (Precision) Exception (#P).”
- SIMD floating-point exceptions; Section 11.5.2.3, “Divide-By-Zero Exception (#Z).”
- Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
- Section 14.10, “SIMD Floating-Point Exceptions.”

4.9.2 Floating-Point Exception Priority

The processor handles exceptions according to a predetermined precedence. When an instruction generates two or more exception conditions, the exception precedence sometimes results in the higher-priority exception being handled and the lower-priority exceptions being ignored. For example, dividing an SNaN by zero can potentially signal an invalid-operation exception (due to the SNaN operand) and a divide-by-zero exception. Here, if both exceptions are masked, the processor handles the higher-priority exception only (the invalid-operation exception), returning a QNaN to the destination. Alternately, a denormal-operand or inexact-result exception can accompany a numeric underflow or overflow exception with both exceptions being handled.

The precedence for floating-point exceptions is as follows:

1. Invalid-operation exception, subdivided as follows:
   a. Stack underflow (occurs with x87 FPU only).
   b. Stack overflow (occurs with x87 FPU only).
   c. Operand of unsupported format (occurs with x87 FPU only when using the double extended precision floating-point format).
   d. SNaN operand.
2. QNaN operand. Though this is not an exception, the handling of a QNaN operand has precedence over lower-priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a zero-divide exception.
3. Any other invalid-operation exception not mentioned above or a divide-by-zero exception.
4. Denormal-operand exception. If masked, then instruction execution continues, and a lower-priority exception can occur as well.
5. Numeric overflow and underflow exceptions; possibly in conjunction with the inexact-result exception.
6. Inexact-result exception.

Invalid operation, zero divide, and denormal operand exceptions are detected before a floating-point operation begins. Overflow, underflow, and precision exceptions are not detected until a true result has been computed. When an unmasked **pre-operation** exception is detected, the destination operand has not yet been updated, and appears as if the offending instruction has not been executed. When an unmasked **post-operation** exception is detected, the destination operand may be updated with a result, depending on the nature of the exception (except for Intel SSE/SSE2/SSE3/AVX instructions, which do not update their destination operands in such cases).

4.9.3 Typical Actions of a Floating-Point Exception Handler

After the floating-point exception handler is invoked, the processor handles the exception in the same manner that it handles non-floating-point exceptions. The floating-point exception handler is normally part of the operating system or executive software, and it usually invokes a user-registered floating-point exception handle.

A typical action of the exception handler is to store state information in memory. Other typical exception handler actions include:

- Examining the stored state information to determine the nature of the error.
- Taking actions to correct the condition that caused the error.
- Clearing the exception flags.
- Returning to the interrupted program and resuming normal execution.

In lieu of writing recovery procedures, the exception handler can do the following:

- Increment in software an exception counter for later display or printing.
- Print or display diagnostic information (such as the state information).
- Halt further program execution.

2. Updates to Chapter 5, Volume 1

Change bars and violet text show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1:* Basic Architecture.

Changes to this chapter:

- Removed note to verify supported vector options in Section 5.31, “Intel® Advanced Vector Extensions 10 Version 1 Instructions.”

CHAPTER 5
INSTRUCTION SET SUMMARY

This chapter provides an abridged overview of Intel 64 and IA-32 instructions. Instructions are divided into the following groups:

- Section 5.1, "General-Purpose Instructions."
- Section 5.2, "x87 FPU Instructions."
- Section 5.3, "x87 FPU AND SIMD State Management Instructions."
- Section 5.4, "MMX Instructions."
- Section 5.5, "Intel® SSE Instructions."
- Section 5.6, "Intel® SSE2 Instructions."
- Section 5.7, "Intel® SSE3 Instructions."
- Section 5.8, "Supplemental Streaming SIMD Extensions 3 (SSSE3) Instructions."
- Section 5.9, "Intel® SSE4 Instructions."
- Section 5.10, "Intel® SSE4.1 Instructions."
- Section 5.11, "Intel® SSE4.2 Instruction Set."
- Section 5.12, "Intel® AES-NI and PCLMULQDQ."
- Section 5.13, "Intel® Advanced Vector Extensions (Intel® AVX)."
- Section 5.14, "16-bit Floating-Point Conversion."
- Section 5.15, "Fused-Multiply-ADD (FMA)."
- Section 5.16, "Intel® Advanced Vector Extensions 2 (Intel® AVX2)."
- Section 5.17, "Intel® Transactional Synchronization Extensions (Intel® TSX)."
- Section 5.18, "Intel® SHA Extensions."
- Section 5.19, "Intel® Advanced Vector Extensions 512 (Intel® AVX-512)."
- Section 5.20, "System Instructions."
- Section 5.21, "64-Bit Mode Instructions."
- Section 5.22, "Virtual-Machine Extensions."
- Section 5.23, "Safer Mode Extensions."
- Section 5.24, "Intel® Memory Protection Extensions."
- Section 5.25, "Intel® Software Guard Extensions."
- Section 5.26, "Shadow Stack Management Instructions."
- Section 5.27, "Control Transfer Terminating Instructions."
- Section 5.28, "Intel® AMX Instructions."
- Section 5.29, "User Interrupt Instructions."
- Section 5.30, "Enqueue Store Instructions."
- Section 5.31, "Intel® Advanced Vector Extensions 10 Version 1 Instructions."

Table 5-1 lists the groups and IA-32 processors that support each group. More recent instruction set extensions are listed in Table 5-2. Within these groups, most instructions are collected into functional subgroups.

Table 5-1. Instruction Groups in Intel® 64 and IA-32 Processors

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors.
X87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
X87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90 nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxxx, 5xxx, 7xxx Series, Intel Atom processors.
SSSE3 Extensions	Intel Xeon processor 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors.
IA-32e mode: 64-bit mode instructions	Intel 64 processors.
System Instructions	Intel 64 and IA-32 processors.
VMX Instructions	Intel 64 and IA-32 processors supporting Intel Virtualization Technology.
SMX Instructions	Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx.

Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors

Instruction Set Architecture	Processor Generation Introduction
SSE4.1 Extensions	Intel® Xeon® processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel® Core™ 2 Extreme processors QX9000 series, Intel® Core™ 2 Quad processor Q9000 series, Intel® Core™ 2 Duo processors 8000 series and T9000 series, Intel Atom® processor based on Silvermont microarchitecture.
SSE4.2 Extensions, CRC32, POPCNT	Intel® Core™ i7 965 processor, Intel® Xeon® processors X3400, X3500, X5500, X6500, X7500 series, Intel Atom processor based on Silvermont microarchitecture.
Intel® AES-NI, PCLMULQDQ	Intel® Xeon® processor E7 series, Intel® Xeon® processors X3600 and X5600, Intel® Core™ i7 980X processor, Intel Atom processor based on Silvermont microarchitecture. Use CPUID to verify presence of Intel AES-NI and PCLMULQDQ across Intel® Core™ processor families.
Intel® AVX	Intel® Xeon® processor E3 and E5 families, 2nd Generation Intel® Core™ i7, i5, i3 processor 2xxx families.
F16C	3rd Generation Intel® Core™ processors, Intel® Xeon® processor E3-1200 v2 product family, Intel® Xeon® processor E5 v2 and E7 v2 families.
RDRAND	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom processor based on Silvermont microarchitecture.
FS/GS base access	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom® processor based on Goldmont microarchitecture.

Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors (Contd.)

Instruction Set Architecture	Processor Generation Introduction
FMA, AVX2, BMI1, BMI2, INVPCID, LZCNT, Intel® TSX	Intel® Xeon® processor E3/E5/E7 v3 product families, 4th Generation Intel® Core™ processor family.
MOVBE	Intel Xeon processor E3/E5/E7 v3 product families, 4th Generation Intel Core processor family, Intel Atom processors.
PREFETCHW	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family, Intel Atom processor based on Silvermont microarchitecture.
ADX	Intel Core M processor family, 5th Generation Intel Core processor family.
RDSEED, CLAC, STAC	Intel Core M processor family, 5th Generation Intel Core processor family, Intel Atom processor based on Goldmont microarchitecture.
AVX512ER, AVX512PF, PREFETCHWT1	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series.
AVX512F, AVX512CD	Intel Xeon Phi Processor 3200, 5200, 7200 Series, Intel® Xeon® Scalable Processor Family, Intel® Core™ i3-8121U processor.
CLFLUSHOPT, XSAVEC, XSAVES, Intel® MPX	Intel Xeon Scalable Processor Family, 6th Generation Intel® Core™ processor family, Intel Atom processor based on Goldmont microarchitecture.
SGX1	6th Generation Intel Core processor family, Intel Atom® processor based on Goldmont Plus microarchitecture.
AVX512DQ, AVX512BW, AVX512VL	Intel Xeon Scalable Processor Family, Intel Core i3-8121U processor based on Cannon Lake microarchitecture.
CLWB	Intel Xeon Scalable Processor Family, Intel Atom® processor based on Tremont microarchitecture, 11th Generation Intel Core processor family based on Tiger Lake microarchitecture.
PKU	Intel Xeon Scalable Processor Family, 10th generation Intel® Core™ processors based on Comet Lake microarchitecture.
AVX512_IFMA, AVX512_VBMI	Intel Core i3-8121U processor based on Cannon Lake microarchitecture.
Intel® SHA Extensions	Intel Core i3-8121U processor based on Cannon Lake microarchitecture, Intel Atom processor based on Goldmont microarchitecture, 3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.
UMIP	Intel Core i3-8121U processor based on Cannon Lake microarchitecture, Intel Atom processor based on Goldmont Plus microarchitecture.
PTWRITE	Intel Atom processor based on Goldmont Plus microarchitecture, 12th generation Intel® Core™ processor supporting Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
RDPID	10th Generation Intel® Core™ processor family based on Ice Lake microarchitecture, Intel Atom processor based on Goldmont Plus microarchitecture.
AVX512_4FMAPS, AVX512_4VNNIW	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series.
AVX512_VNNI	2nd Generation Intel® Xeon® Scalable Processor Family, 10th Generation Intel Core processor family based on Ice Lake microarchitecture.
AVX512_VPOPCNTDQ	Intel Xeon Phi Processor 7215, 7285, 7295 Series, 10th Generation Intel Core processor family based on Ice Lake microarchitecture.
Fast Short REP MOV	10th Generation Intel Core processor family based on Ice Lake microarchitecture.
GFNI (SSE)	10th Generation Intel Core processor family based on Ice Lake microarchitecture, Intel Atom processor based on Tremont microarchitecture.

Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors (Contd.)

Instruction Set Architecture	Processor Generation Introduction
VAES, GFNI (AVX/AVX512), AVX512_VBMI2, VPCLMULQDQ, AVX512_BITALG	10th Generation Intel Core processor family based on Ice Lake microarchitecture.
ENCLV	Future processors.
Split Lock Detection	10th Generation Intel Core processor family based on Ice Lake microarchitecture, Intel Atom processor based on Tremont microarchitecture.
CLDEMOTE	Intel Atom processor based on Tremont microarchitecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
Direct stores: MOVDIRI, MOVDIR64B	Intel Atom processor based on Tremont microarchitecture, 11th Generation Intel Core processor family based on Tiger Lake microarchitecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
User wait: TPAUSE, UMONITOR, UMWAIT	Intel Atom processor based on Tremont microarchitecture, 12th generation Intel Core processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
AVX512-BF16	3rd Generation Intel® Xeon® Scalable Processor Family based on Cooper Lake product, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
AVX512_VP2INTERSECT	11th Generation Intel Core processor family based on Tiger Lake microarchitecture. (Not currently supported in any other processors).
Key Locker[1]	11th Generation Intel Core processor family based on Tiger Lake microarchitecture, 12th generation Intel Core processor supporting Alder Lake performance hybrid architecture.
Control-flow Enforcement Technology (CET)	11th Generation Intel Core processor family based on Tiger Lake microarchitecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
TME-MK[2], PCONFIG	3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.
WBNOINVD	3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.
LBRs (architectural)	12th generation Intel Core processor supporting Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
Intel® Virtualization Technology - Redirect Protection (Intel® VT-rp) and HLAT	12th generation Intel Core processor supporting Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
AVX-VNNI	12th generation Intel Core processor supporting Alder Lake performance hybrid architecture[3], 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
SERIALIZE	12th generation Intel Core processor supporting Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
Intel® Thread Director and HRESET	12th generation Intel Core processor supporting Alder Lake performance hybrid architecture.
Fast zero-length REP MOVSB, fast short REP STOSB	12th generation Intel Core processor supporting Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
Fast Short REP CMPSB, fast short REP SCASB	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.

Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors (Contd.)

Instruction Set Architecture	Processor Generation Introduction
Supervisor Memory Protection Keys (PKS)	12th generation Intel Core processor supporting Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
Attestation Services for Intel® SGX	3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.
Enqueue Stores: ENQCMD and ENQCMDS	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
Intel® TSX Suspend Load Address Tracking (TSXLDTRK)	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
Intel® Advanced Matrix Extensions (Intel® AMX) Includes CPUID Leaf 1EH, “TMUL Information Main Leaf”, and CPUID bits AMX-BF16, AMX-TILE, and AMX-INT8.	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
User Interrupts (UINTR)	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
IPI Virtualization	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
AVX512-FP16, for the FP16 Data Type	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
Virtualization of guest accesses to IA32_SPEC_CTRL	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
Linear Address Masking (LAM)	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
Linear Address Space Separation (LASS)	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
PREFETCHIT0/1	Intel® Xeon® 6 P-core processors based on Granite Rapids microarchitecture.
AMX-FP16	Intel® Xeon® 6 P-core processors based on Granite Rapids microarchitecture.
CMPCCXADD	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
AVX-IFMA	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
AVX-NE-CONVERT	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
AVX-VNNI-INT8	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
AVX-VNNI-INT16	Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
SHA512	Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
SM3	Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
SM4	Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.

Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors (Contd.)

Instruction Set Architecture	Processor Generation Introduction
RDMSRLIST, WRMSRLIST, and WRMSRNS	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
UC Lock Disable Causes #AC	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture.
LBR Event Logging	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
UIRET flexibly updates UIF	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture, Intel® Core™ Ultra processor supporting Lunar Lake performance hybrid architecture.
Intel® Advanced Vector Extensions 10 Version 1 (Intel® AVX10.1)	Intel® Xeon® 6 P-core processors based on Granite Rapids microarchitecture.

NOTES:

1. Details on Key Locker can be found in the Intel Key Locker Specification here:
   https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.
2. Further details on TME-MK usage can be found here:
   https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf.
3. Alder Lake performance hybrid architecture does not support Intel® AVX-512. ISA features such as Intel® AVX, AVX-VNNI, Intel® AVX2, and UMONITOR/UMWAIT/TPAUSE are supported.

The following sections list instructions in each major group and subgroup. Given for each instruction is its mnemonic and descriptive names. When two or more mnemonics are given (for example, CMOVA/CMOVNBE), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMOVA (Conditional move if above) and CMOVNBE (Conditional move if not below or equal) represent the same condition. For detailed information about specific instructions, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

5.1 GENERAL-PURPOSE INSTRUCTIONS

The general-purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operations that programmers commonly use to write application and system software to run on Intel 64 and IA-32 processors. They operate on data contained in memory, in the general-purpose registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP) and in the EFLAGS register. They also operate on address information contained in memory, the general-purpose registers, and the segment registers (CS, DS, SS, ES, FS, and GS).

This group of instructions includes the data transfer, binary integer arithmetic, decimal arithmetic, logic operations, shift and rotate, bit and byte operations, program control, string, flag control, segment register operations, and miscellaneous subgroups. The sections that follow introduce each subgroup.

For more detailed information on general purpose-instructions, see Chapter 7, "Programming With General-Purpose Instructions."

5.1.1 Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers.
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero.
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero.

CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal.
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below.
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal.
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above.
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal.
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less.
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal.
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater.
CMOVC	Conditional move if carry.
CMOVNC	Conditional move if not carry.
CMOVO	Conditional move if overflow.
CMOVNO	Conditional move if not overflow.
CMOVS	Conditional move if sign (negative).
CMOVNS	Conditional move if not sign (non-negative).
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even.
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd.
XCHG	Exchange.
BSWAP	Byte swap.
XADD	Exchange and add.
CMPXCHG	Compare and exchange.
CMPXCHG8B	Compare and exchange 8 bytes.
PUSH	Push onto stack.
POP	Pop off of stack.
PUSHA/PUSHAD	Push general-purpose registers onto stack.
POPA/POPAD	Pop general-purpose registers from stack.
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword.
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register.
MOVSX	Move and sign extend.
MOVZX	Move and zero extend.

5.1.2 Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

ADCX	Unsigned integer add with carry.
ADOX	Unsigned integer add with overflow.
ADD	Integer add.
ADC	Add with carry.
SUB	Subtract.
SBB	Subtract with borrow.
IMUL	Signed multiply.
MUL	Unsigned multiply.
IDIV	Signed divide.
DIV	Unsigned divide.
INC	Increment.
DEC	Decrement.
NEG	Negate.

CMP	Compare.

5.1.3 Decimal Arithmetic Instructions

The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.

DAA	Decimal adjust after addition.
DAS	Decimal adjust after subtraction.
AAA	ASCII adjust after addition.
AAS	ASCII adjust after subtraction.
AAM	ASCII adjust after multiplication.
AAD	ASCII adjust before division.

5.1.4 Logical Instructions

The logical instructions perform basic AND, OR, XOR, and NOT logical operations on byte, word, and doubleword values.

AND	Perform bitwise logical AND.
OR	Perform bitwise logical OR.
XOR	Perform bitwise logical exclusive OR.
NOT	Perform bitwise logical NOT.

5.1.5 Shift and Rotate Instructions

The shift and rotate instructions shift and rotate the bits in word and doubleword operands.

SAR	Shift arithmetic right.
SHR	Shift logical right.
SAL/SHL	Shift arithmetic left/Shift logical left.
SHRD	Shift right double.
SHLD	Shift left double.
ROR	Rotate right.
ROL	Rotate left.
RCR	Rotate through carry right.
RCL	Rotate through carry left.

5.1.6 Bit and Byte Instructions

Bit instructions test and modify individual bits in word and doubleword operands. Byte instructions set the value of a byte operand to indicate the status of flags in the EFLAGS register.

BT	Bit test.
BTS	Bit test and set.
BTR	Bit test and reset.
BTC	Bit test and complement.
BSF	Bit scan forward.
BSR	Bit scan reverse.
SETE/SETZ	Set byte if equal/Set byte if zero.
SETNE/SETNZ	Set byte if not equal/Set byte if not zero.
SETA/SETNBE	Set byte if above/Set byte if not below or equal.

SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry.
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry.
SETBE/SETNA	Set byte if below or equal/Set byte if not above.
SETG/SETNLE	Set byte if greater/Set byte if not less or equal.
SETGE/SETNL	Set byte if greater or equal/Set byte if not less.
SETL/SETNGE	Set byte if less/Set byte if not greater or equal.
SETLE/SETNG	Set byte if less or equal/Set byte if not greater.
SETS	Set byte if sign (negative).
SETNS	Set byte if not sign (non-negative).
SETO	Set byte if overflow.
SETNO	Set byte if not overflow.
SETPE/SETP	Set byte if parity even/Set byte if parity.
SETPO/SETNP	Set byte if parity odd/Set byte if not parity.
TEST	Logical compare.
CRC32[1]	Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.
POPCNT[2]	Calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

5.1.7 Control Transfer Instructions

The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

JMP	Jump.
JE/JZ	Jump if equal/Jump if zero.
JNE/JNZ	Jump if not equal/Jump if not zero.
JA/JNBE	Jump if above/Jump if not below or equal.
JAE/JNB	Jump if above or equal/Jump if not below.
JB/JNAE	Jump if below/Jump if not above or equal.
JBE/JNA	Jump if below or equal/Jump if not above.
JG/JNLE	Jump if greater/Jump if not less or equal.
JGE/JNL	Jump if greater or equal/Jump if not less.
JL/JNGE	Jump if less/Jump if not greater or equal.
JLE/JNG	Jump if less or equal/Jump if not greater.
JC	Jump if carry.
JNC	Jump if not carry.
JO	Jump if overflow.
JNO	Jump if not overflow.
JS	Jump if sign (negative).
JNS	Jump if not sign (non-negative).
JPO/JNP	Jump if parity odd/Jump if not parity.
JPE/JP	Jump if parity even/Jump if parity.
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero.
LOOP	Loop with ECX counter.

1. Processor support of CRC32 is enumerated by CPUID.01:ECX[SSE4.2] = 1
2. Processor support of POPCNT is enumerated by CPUID.01:ECX[POPCNT] = 1

LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal.
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal.
CALL	Call procedure.
RET	Return.
IRET	Return from interrupt.
INT	Software interrupt.
INTO	Interrupt on overflow.
BOUND	Detect value out of range.
ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

5.1.8 String Instructions

The string instructions operate on strings of bytes, allowing them to be moved to and from memory.

MOVS/MOVSB	Move string/Move byte string.
MOVS/MOVSW	Move string/Move word string.
MOVS/MOVSD	Move string/Move doubleword string.
CMPS/CMPSB	Compare string/Compare byte string.
CMPS/CMPSW	Compare string/Compare word string.
CMPS/CMPSD	Compare string/Compare doubleword string.
SCAS/SCASB	Scan string/Scan byte string.
SCAS/SCASW	Scan string/Scan word string.
SCAS/SCASD	Scan string/Scan doubleword string.
LODS/LODSB	Load string/Load byte string.
LODS/LODSW	Load string/Load word string.
LODS/LODSD	Load string/Load doubleword string.
STOS/STOSB	Store string/Store byte string.
STOS/STOSW	Store string/Store word string.
STOS/STOSD	Store string/Store doubleword string.
REP	Repeat while ECX not zero.
REPE/REPZ	Repeat while equal/Repeat while zero.
REPNE/REPNZ	Repeat while not equal/Repeat while not zero.

5.1.9 I/O Instructions

These instructions move data between the processor’s I/O ports and a register or memory.

IN	Read from a port.
OUT	Write to a port.
INS/INSB	Input string from port/Input byte string from port.
INS/INSW	Input string from port/Input word string from port.
INS/INSD	Input string from port/Input doubleword string from port.
OUTS/OUTSB	Output string to port/Output byte string to port.
OUTS/OUTSW	Output string to port/Output word string to port.
OUTS/OUTSD	Output string to port/Output doubleword string to port.

5.1.10 Enter and Leave Instructions

These instructions provide machine-language support for procedure calls in block-structured languages.

ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

5.1.11 Flag Control (EFLAG) Instructions

The flag control instructions operate on the flags in the EFLAGS register.

STC	Set carry flag.
CLC	Clear the carry flag.
CMC	Complement the carry flag.
CLD	Clear the direction flag.
STD	Set direction flag.
LAHF	Load flags into AH register.
SAHF	Store AH register into flags.
PUSHF/PUSHFD	Push EFLAGS onto stack.
POPF/POPFD	Pop EFLAGS from stack.
STI	Set interrupt flag.
CLI	Clear the interrupt flag.

5.1.12 Segment Register Instructions

The segment register instructions allow far pointers (segment addresses) to be loaded into the segment registers.

LDS	Load far pointer using DS.
LES	Load far pointer using ES.
LFS	Load far pointer using FS.
LGS	Load far pointer using GS.
LSS	Load far pointer using SS.

5.1.13 Miscellaneous Instructions

The miscellaneous instructions provide such functions as loading an effective address, executing a “no-operation,” and retrieving processor identification information.

LEA	Load effective address.
NOP	No operation.
UD	Undefined instruction.
XLAT/XLATB	Table lookup translation.
CPUID	Processor identification.
MOVBE[1]	Move data after swapping data bytes.
PREFETCHW	Prefetch data into cache in anticipation of write.
PREFETCHWT1	Prefetch hint T1 with intent to write.
CLFLUSH	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor’s cache hierarchy.
CLFLUSHOPT	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor’s cache hierarchy with optimized memory system throughput.

1. Processor support of MOVBE is enumerated by CPUID.01:ECX.MOVBE[bit 22] = 1.

5.1.14 User Mode Extended State Save/Restore Instructions

XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XRSTOR	Restore processor extended states from memory.
XGETBV	Reads the state of an extended control register.

5.1.15 Random Number Generator Instructions

RDRAND	Retrieves a random number generated from hardware.
RDSEED	Retrieves a random number generated from hardware.

5.1.16 BMI1 and BMI2 Instructions

ANDN	Bitwise AND of first source with inverted second source operands.
BEXTR	Contiguous bitwise extract.
BLSI	Extract lowest set bit.
BLSMSK	Set all lower bits below first set bit to 1.
BLSR	Reset lowest set bit.
BZHI	Zero high bits starting from specified bit position.
LZCNT	Count the number of leading zero bits.
MULX	Unsigned multiply without affecting arithmetic flags.
PDEP	Parallel deposit of bits using a mask.
PEXT	Parallel extraction of bits using a mask.
RORX	Rotate right without affecting arithmetic flags.
SARX	Shift arithmetic right.
SHLX	Shift logic left.
SHRX	Shift logic right.
TZCNT	Count the number of trailing zero bits.

5.1.16.1 Detection of VEX-Encoded GPR Instructions, LZCNT, TZCNT, and PREFETCHW

VEX-encoded general-purpose instructions do not operate on any vector registers.

There are separate feature flags for the following subsets of instructions that operate on general purpose registers, and the detection requirements for hardware support are:

CPUID.(EAX=07H, ECX=0H):EBX.BMI1[bit 3]: if 1 indicates the processor supports the first group of advanced bit manipulation extensions (ANDN, BEXTR, BLSI, BLSMSK, BLSR, TZCNT);

CPUID.(EAX=07H, ECX=0H):EBX.BMI2[bit 8]: if 1 indicates the processor supports the second group of advanced bit manipulation extensions (BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX);

CPUID.EAX=80000001H:ECX.LZCNT[bit 5]: if 1 indicates the processor supports the LZCNT instruction.

CPUID.EAX=80000001H:ECX.PREFTEHCHW[bit 8]: if 1 indicates the processor supports the PREFTEHCHW instruction. CPUID.(EAX=07H, ECX=0H):ECX.PREFTEHCHWT1[bit 0]: if 1 indicates the processor supports the PREFTEHCHWT1 instruction.

5.2 X87 FPU INSTRUCTIONS

The x87 FPU instructions are executed by the processor's x87 FPU. These instructions operate on floating-point, integer, and binary-coded decimal (BCD) operands. For more detail on x87 FPU instructions, see Chapter 8, "Programming with the x87 FPU."

These instructions are divided into the following subgroups: data transfer, load constants, and FPU control instructions. The sections that follow introduce each subgroup.

5.2.1 X87 FPU Data Transfer Instructions

The data transfer instructions move floating-point, integer, and BCD values between memory and the x87 FPU registers. They also perform conditional move operations on floating-point operands.

FLD	Load floating-point value.
FST	Store floating-point value.
FSTP	Store floating-point value and pop.
FILD	Load integer.
FIST	Store integer.
FISTP[1]	Store integer and pop.
FBLD	Load BCD.
FBSTP	Store BCD and pop.
FXCH	Exchange registers.
FCMOVE	Floating-point conditional move if equal.
FCMOVNE	Floating-point conditional move if not equal.
FCMOVB	Floating-point conditional move if below.
FCMOVBE	Floating-point conditional move if below or equal.
FCMOVNB	Floating-point conditional move if not below.
FCMOVNBE	Floating-point conditional move if not below or equal.
FCMOVU	Floating-point conditional move if unordered.
FCMOVNU	Floating-point conditional move if not unordered.

5.2.2 X87 FPU Basic Arithmetic Instructions

The basic arithmetic instructions perform basic arithmetic operations on floating-point and integer operands.

FADD	Add floating-point.
FADDP	Add floating-point and pop.
FIADD	Add integer.
FSUB	Subtract floating-point.
FSUBP	Subtract floating-point and pop.
FISUB	Subtract integer.
FSUBR	Subtract floating-point reverse.
FSUBRP	Subtract floating-point reverse and pop.
FISUBR	Subtract integer reverse.
FMUL	Multiply floating-point.
FMULP	Multiply floating-point and pop.
FIMUL	Multiply integer.
FDIV	Divide floating-point.

1. SSE3 provides an instruction FISTTP for integer conversion.

FDIVP	Divide floating-point and pop.
FIDIV	Divide integer.
FDIVR	Divide floating-point reverse.
FDIVRP	Divide floating-point reverse and pop.
FIDIVR	Divide integer reverse.
FPREM	Partial remainder.
FPREM1	IEEE partial remainder.
FABS	Absolute value.
FCHS	Change sign.
FRNDINT	Round to integer.
FSCALE	Scale by power of two.
FSQRT	Square root.
FXTRACT	Extract exponent and significand.

5.2.3 X87 FPU Comparison Instructions

The compare instructions examine or compare floating-point or integer operands.

FCOM	Compare floating-point.
FCOMP	Compare floating-point and pop.
FCOMPP	Compare floating-point and pop twice.
FUCOM	Unordered compare floating-point.
FUCOMP	Unordered compare floating-point and pop.
FUCOMPP	Unordered compare floating-point and pop twice.
FICOM	Compare integer.
FICOMP	Compare integer and pop.
FCOMI	Compare floating-point and set EFLAGS.
FUCOMI	Unordered compare floating-point and set EFLAGS.
FCOMIP	Compare floating-point, set EFLAGS, and pop.
FUCOMIP	Unordered compare floating-point, set EFLAGS, and pop.
FTST	Test floating-point (compare with 0.0).
FXAM	Examine floating-point.

5.2.4 X87 FPU Transcendental Instructions

The transcendental instructions perform basic trigonometric and logarithmic operations on floating-point operands.

FSIN	Sine.
FCOS	Cosine.
FSINCOS	Sine and cosine.
FPTAN	Partial tangent.
FPATAN	Partial arctangent.
F2XM1	$2^x - 1$.
FYL2X	$y * \log_2 x$.
FYL2XP1	$y * \log_2(x+1)$.

5.2.5 X87 FPU Load Constants Instructions

The load constants instructions load common constants, such as π, into the x87 floating-point registers.

FLD1	Load +1.0.
FLDZ	Load +0.0.
FLDPI	Load π.
FLDL2E	Load $\log_2 e$.
FLDLN2	Load $\log_e 2$.
FLDL2T	Load $\log_2 10$.
FLDLG2	Load $\log_{10} 2$.

5.2.6 X87 FPU Control Instructions

The x87 FPU control instructions operate on the x87 FPU register stack and save and restore the x87 FPU state.

FINCSTP	Increment FPU register stack pointer.
FDECSTP	Decrement FPU register stack pointer.
FFREE	Free floating-point register.
FINIT	Initialize FPU after checking error conditions.
FNINIT	Initialize FPU without checking error conditions.
FCLEX	Clear floating-point exception flags after checking for error conditions.
FNCLEX	Clear floating-point exception flags without checking for error conditions.
FSTCW	Store FPU control word after checking error conditions.
FNSTCW	Store FPU control word without checking error conditions.
FLDCW	Load FPU control word.
FSTENV	Store FPU environment after checking error conditions.
FNSTENV	Store FPU environment without checking error conditions.
FLDENV	Load FPU environment.
FSAVE	Save FPU state after checking error conditions.
FNSAVE	Save FPU state without checking error conditions.
FRSTOR	Restore FPU state.
FSTSW	Store FPU status word after checking error conditions.
FNSTSW	Store FPU status word without checking error conditions.
WAIT/FWAIT	Wait for FPU.
FNOP	FPU no operation.

5.3 X87 FPU AND SIMD STATE MANAGEMENT INSTRUCTIONS

Two state management instructions were introduced into the IA-32 architecture with the Pentium II processor family:

FXSAVE	Save x87 FPU and SIMD state.
FXRSTOR	Restore x87 FPU and SIMD state.

Initially, these instructions operated only on the x87 FPU (and MMX) registers to perform a fast save and restore, respectively, of the x87 FPU and MMX state. With the introduction of SSE extensions in the Pentium III processor family, these instructions were expanded to also save and restore the state of the XMM and MXCSR registers. Intel 64 architecture also supports these instructions.

See Section 10.5, “FXSAVE and FXRSTOR Instructions,” for more detail.

5.4 MMX INSTRUCTIONS

Four extensions have been introduced into the IA-32 architecture to permit IA-32 processors to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2 extensions, and SSE3 extensions. For a discussion that puts SIMD instructions in their historical context, see Section 2.2.7, "SIMD Instructions."

MMX instructions operate on packed byte, word, doubleword, or quadword integer operands contained in memory, in MMX registers, and/or in general-purpose registers. For more detail on these instructions, see Chapter 9, "Programming with Intel® MMX™ Technology."

MMX instructions can only be executed on Intel 64 and IA-32 processors that support the MMX technology. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

MMX instructions are divided into the following subgroups: data transfer, conversion, packed arithmetic, comparison, logical, shift and rotate, and state management instructions. The sections that follow introduce each subgroup.

5.4.1 MMX Data Transfer Instructions

The data transfer instructions move doubleword and quadword operands between MMX registers and between MMX registers and memory.

MOVD	Move doubleword.
MOVQ	Move quadword.

5.4.2 MMX Conversion Instructions

The conversion instructions pack and unpack bytes, words, and doublewords

PACKSSWB	Pack words into bytes with signed saturation.
PACKSSDW	Pack doublewords into words with signed saturation.
PACKUSWB	Pack words into bytes with unsigned saturation.
PUNPCKHBW	Unpack high-order bytes.
PUNPCKHWD	Unpack high-order words.
PUNPCKHDQ	Unpack high-order doublewords.
PUNPCKLBW	Unpack low-order bytes.
PUNPCKLWD	Unpack low-order words.
PUNPCKLDQ	Unpack low-order doublewords.

5.4.3 MMX Packed Arithmetic Instructions

The packed arithmetic instructions perform packed integer arithmetic on packed byte, word, and doubleword integers.

PADDB	Add packed byte integers.
PADDW	Add packed word integers.
PADDD	Add packed doubleword integers.
PADDSB	Add packed signed byte integers with signed saturation.
PADDSW	Add packed signed word integers with signed saturation.
PADDUSB	Add packed unsigned byte integers with unsigned saturation.
PADDUSW	Add packed unsigned word integers with unsigned saturation.
PSUBB	Subtract packed byte integers.
PSUBW	Subtract packed word integers.
PSUBD	Subtract packed doubleword integers.
PSUBSB	Subtract packed signed byte integers with signed saturation.
PSUBSW	Subtract packed signed word integers with signed saturation.

PSUBUSB	Subtract packed unsigned byte integers with unsigned saturation.
PSUBUSW	Subtract packed unsigned word integers with unsigned saturation.
PMULHW	Multiply packed signed word integers and store high result.
PMULLW	Multiply packed signed word integers and store low result.
PMADDWD	Multiply and add packed word integers.

5.4.4 MMX Comparison Instructions

The compare instructions compare packed bytes, words, or doublewords.

PCMPEQB	Compare packed bytes for equal.
PCMPEQW	Compare packed words for equal.
PCMPEQD	Compare packed doublewords for equal.
PCMPGTB	Compare packed signed byte integers for greater than.
PCMPGTW	Compare packed signed word integers for greater than.
PCMPGTD	Compare packed signed doubleword integers for greater than.

5.4.5 MMX Logical Instructions

The logical instructions perform AND, AND NOT, OR, and XOR operations on quadword operands.

PAND	Bitwise logical AND.
PANDN	Bitwise logical AND NOT.
POR	Bitwise logical OR.
PXOR	Bitwise logical exclusive OR.

5.4.6 MMX Shift and Rotate Instructions

The shift and rotate instructions shift and rotate packed bytes, words, or doublewords, or quadwords in 64-bit operands.

PSLLW	Shift packed words left logical.
PSLLD	Shift packed doublewords left logical.
PSLLQ	Shift packed quadword left logical.
PSRLW	Shift packed words right logical.
PSRLD	Shift packed doublewords right logical.
PSRLQ	Shift packed quadword right logical.
PSRAW	Shift packed words right arithmetic.
PSRAD	Shift packed doublewords right arithmetic.

5.4.7 MMX State Management Instructions

The EMMS instruction clears the MMX state from the MMX registers.

EMMS	Empty MMX state.

5.5 INTEL® SSE INSTRUCTIONS

Intel SSE instructions represent an extension of the SIMD execution model introduced with the MMX technology. For more detail on these instructions, see Chapter 10, “Programming with Intel® Streaming SIMD Extensions (Intel® SSE).”

Intel SSE instructions can only be executed on Intel 64 and IA-32 processors that support Intel SSE extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

Intel SSE instructions are divided into four subgroups (note that the first subgroup has subordinate subgroups of its own):

- SIMD single precision floating-point instructions that operate on the XMM registers.
- MXCSR state management instructions.
- 64-bit SIMD integer instructions that operate on the MMX registers.
- Cacheability control, prefetch, and instruction ordering instructions.

The following sections provide an overview of these groups.

5.5.1 Intel® SSE SIMD Single Precision Floating-Point Instructions

These instructions operate on packed and scalar single precision floating-point values located in XMM registers and/or memory. This subgroup is further divided into the following subordinate subgroups: data transfer, packed arithmetic, comparison, logical, shuffle and unpack, and conversion instructions.

5.5.1.1 Intel® SSE Data Transfer Instructions

Intel SSE data transfer instructions move packed and scalar single precision floating-point operands between XMM registers and between XMM registers and memory.

MOVAPS	Move four aligned packed single precision floating-point values between XMM registers or between an XMM register and memory.
MOVUPS	Move four unaligned packed single precision floating-point values between XMM registers or between an XMM register and memory.
MOVHPS	Move two packed single precision floating-point values to and from the high quadword of an XMM register and memory.
MOVHLPS	Move two packed single precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register.
MOVLPS	Move two packed single precision floating-point values to and from the low quadword of an XMM register and memory.
MOVLHPS	Move two packed single precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register.
MOVMSKPS	Extract sign mask from four packed single precision floating-point values.
MOVSS	Move scalar single precision floating-point value between XMM registers or between an XMM register and memory.

5.5.1.2 Intel® SSE Packed Arithmetic Instructions

Intel SSE packed arithmetic instructions perform packed and scalar arithmetic operations on packed and scalar single precision floating-point operands.

ADDPS	Add packed single precision floating-point values.
ADDSS	Add scalar single precision floating-point values.
SUBPS	Subtract packed single precision floating-point values.
SUBSS	Subtract scalar single precision floating-point values.
MULPS	Multiply packed single precision floating-point values.
MULSS	Multiply scalar single precision floating-point values.
DIVPS	Divide packed single precision floating-point values.
DIVSS	Divide scalar single precision floating-point values.

RCPPS	Compute reciprocals of packed single precision floating-point values.
RCPSS	Compute reciprocal of scalar single precision floating-point values.
SQRTPS	Compute square roots of packed single precision floating-point values.
SQRTSS	Compute square root of scalar single precision floating-point values.
RSQRTPS	Compute reciprocals of square roots of packed single precision floating-point values.
RSQRTSS	Compute reciprocal of square root of scalar single precision floating-point values.
MAXPS	Return maximum packed single precision floating-point values.
MAXSS	Return maximum scalar single precision floating-point values.
MINPS	Return minimum packed single precision floating-point values.
MINSS	Return minimum scalar single precision floating-point values.

5.5.1.3 Intel® SSE Comparison Instructions

Intel SSE compare instructions compare packed and scalar single precision floating-point operands.

CMPPS	Compare packed single precision floating-point values.
CMPSS	Compare scalar single precision floating-point values.
COMISS	Perform ordered comparison of scalar single precision floating-point values and set flags in EFLAGS register.
UCOMISS	Perform unordered comparison of scalar single precision floating-point values and set flags in EFLAGS register.

5.5.1.4 Intel® SSE Logical Instructions

Intel SSE logical instructions perform bitwise AND, AND NOT, OR, and XOR operations on packed single precision floating-point operands.

ANDPS	Perform bitwise logical AND of packed single precision floating-point values.
ANDNPS	Perform bitwise logical AND NOT of packed single precision floating-point values.
ORPS	Perform bitwise logical OR of packed single precision floating-point values.
XORPS	Perform bitwise logical XOR of packed single precision floating-point values.

5.5.1.5 Intel® SSE Shuffle and Unpack Instructions

Intel SSE shuffle and unpack instructions shuffle or interleave single precision floating-point values in packed single precision floating-point operands.

SHUFPS	Shuffles values in packed single precision floating-point operands.
UNPCKHPS	Unpacks and interleaves the two high-order values from two single precision floating-point operands.
UNPCKLPS	Unpacks and interleaves the two low-order values from two single precision floating-point operands.

5.5.1.6 Intel® SSE Conversion Instructions

Intel SSE conversion instructions convert packed and individual doubleword integers into packed and scalar single precision floating-point values and vice versa.

CVTPI2PS	Convert packed doubleword integers to packed single precision floating-point values.
CVTSI2SS	Convert signed integer to scalar single precision floating-point value.
CVTPS2PI	Convert packed single precision floating-point values to packed doubleword integers.
CVTTPS2PI	Convert with truncation packed single precision floating-point values to packed doubleword integers.
CVTSS2SI	Convert a scalar single precision floating-point value to a signed integer.

CVTTSS2SI	Convert with truncation a scalar single precision floating-point value to a scalar signed integer.

5.5.2 Intel® SSE MXCSR State Management Instructions

MXCSR state management instructions allow saving and restoring the state of the MXCSR control and status register.

LDMXCSR	Load MXCSR register.
STMXCSR	Save MXCSR register state.

5.5.3 Intel® SSE 64-Bit SIMD Integer Instructions

These Intel SSE 64-bit SIMD integer instructions perform additional operations on packed bytes, words, or doublewords contained in MMX registers. They represent enhancements to the MMX instruction set described in Section 5.4, "MMX Instructions."

PAVGB	Compute average of packed unsigned byte integers.
PAVGW	Compute average of packed unsigned word integers.
PEXTRW	Extract word.
PINSRW	Insert word.
PMAXUB	Maximum of packed unsigned byte integers.
PMAXSW	Maximum of packed signed word integers.
PMINUB	Minimum of packed unsigned byte integers.
PMINSW	Minimum of packed signed word integers.
PMOVMSKB	Move byte mask.
PMULHUW	Multiply packed unsigned integers and store high result.
PSADBW	Compute sum of absolute differences.
PSHUFW	Shuffle packed integer word in MMX register.

5.5.4 Intel® SSE Cacheability Control, Prefetch, and Instruction Ordering Instructions

The cacheability control instructions provide control over the caching of non-temporal data when storing data from the MMX and XMM registers to memory. The PREFETCH*h* allows data to be prefetched to a selected cache level. The SFENCE instruction controls instruction ordering on store operations.

MASKMOVQ	Non-temporal store of selected bytes from an MMX register into memory.
MOVNTQ	Non-temporal store of quadword from an MMX register into memory.
MOVNTPS	Non-temporal store of four packed single precision floating-point values from an XMM register into memory.
PREFETCH*h*	Load 32 or more of bytes from memory to a selected level of the processor's cache hierarchy.
SFENCE	Serializes store operations.

5.6 INTEL® SSE2 INSTRUCTIONS

Intel SSE2 extensions represent an extension of the SIMD execution model introduced with MMX technology and the Intel SSE extensions. Intel SSE2 instructions operate on packed double precision floating-point operands and on packed byte, word, doubleword, and quadword operands located in the XMM registers. For more detail on these instructions, see Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)."

Intel SSE2 instructions can only be executed on Intel 64 and IA-32 processors that support the Intel SSE2 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

These instructions are divided into four subgroups (note that the first subgroup is further divided into subordinate subgroups):

- Packed and scalar double precision floating-point instructions.
- Packed single precision floating-point conversion instructions.
- 128-bit SIMD integer instructions.
- Cacheability-control and instruction ordering instructions.

The following sections give an overview of each subgroup.

5.6.1 Intel® SSE2 Packed and Scalar Double Precision Floating-Point Instructions

Intel SSE2 packed and scalar double precision floating-point instructions are divided into the following subordinate subgroups: data movement, arithmetic, comparison, conversion, logical, and shuffle operations on double precision floating-point operands. These are introduced in the sections that follow.

5.6.1.1 Intel® SSE2 Data Movement Instructions

Intel SSE2 data movement instructions move double precision floating-point data between XMM registers and between XMM registers and memory.

MOVAPD	Move two aligned packed double precision floating-point values between XMM registers or between an XMM register and memory.
MOVUPD	Move two unaligned packed double precision floating-point values between XMM registers or between an XMM register and memory.
MOVHPD	Move high packed double precision floating-point value to and from the high quadword of an XMM register and memory.
MOVLPD	Move low packed single precision floating-point value to and from the low quadword of an XMM register and memory.
MOVMSKPD	Extract sign mask from two packed double precision floating-point values.
MOVSD	Move scalar double precision floating-point value between XMM registers or between an XMM register and memory.

5.6.1.2 Intel® SSE2 Packed Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double precision floating-point operands.

ADDPD	Add packed double precision floating-point values.
ADDSD	Add scalar double precision floating-point values.
SUBPD	Subtract packed double precision floating-point values.
SUBSD	Subtract scalar double precision floating-point values.
MULPD	Multiply packed double precision floating-point values.
MULSD	Multiply scalar double precision floating-point values.
DIVPD	Divide packed double precision floating-point values.
DIVSD	Divide scalar double precision floating-point values.
SQRTPD	Compute packed square roots of packed double precision floating-point values.
SQRTSD	Compute scalar square root of scalar double precision floating-point values.
MAXPD	Return maximum packed double precision floating-point values.
MAXSD	Return maximum scalar double precision floating-point values.
MINPD	Return minimum packed double precision floating-point values.

MINSD	Return minimum scalar double precision floating-point values.

5.6.1.3 Intel® SSE2 Logical Instructions

Intel SSE2 logical instructions perform AND, AND NOT, OR, and XOR operations on packed double precision floating-point values.

ANDPD	Perform bitwise logical AND of packed double precision floating-point values.
ANDNPD	Perform bitwise logical AND NOT of packed double precision floating-point values.
ORPD	Perform bitwise logical OR of packed double precision floating-point values.
XORPD	Perform bitwise logical XOR of packed double precision floating-point values.

5.6.1.4 Intel® SSE2 Compare Instructions

Intel SSE2 compare instructions compare packed and scalar double precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

CMPPD	Compare packed double precision floating-point values.
CMPSD	Compare scalar double precision floating-point values.
COMISD	Perform ordered comparison of scalar double precision floating-point values and set flags in EFLAGS register.
UCOMISD	Perform unordered comparison of scalar double precision floating-point values and set flags in EFLAGS register.

5.6.1.5 Intel® SSE2 Shuffle and Unpack Instructions

Intel SSE2 shuffle and unpack instructions shuffle or interleave double precision floating-point values in packed double precision floating-point operands.

SHUFPD	Shuffles values in packed double precision floating-point operands.
UNPCKHPD	Unpacks and interleaves the high values from two packed double precision floating-point operands.
UNPCKLPD	Unpacks and interleaves the low values from two packed double precision floating-point operands.

5.6.1.6 Intel® SSE2 Conversion Instructions

Intel SSE2 conversion instructions convert packed and individual doubleword integers into packed and scalar double precision floating-point values and vice versa. They also convert between packed and scalar single precision and double precision floating-point values.

CVTPD2PI	Convert packed double precision floating-point values to packed doubleword integers.
CVTTPD2PI	Convert with truncation packed double precision floating-point values to packed doubleword integers.
CVTPI2PD	Convert packed doubleword integers to packed double precision floating-point values.
CVTPD2DQ	Convert packed double precision floating-point values to packed doubleword integers.
CVTTPD2DQ	Convert with truncation packed double precision floating-point values to packed doubleword integers.
CVTDQ2PD	Convert packed doubleword integers to packed double precision floating-point values.
CVTPS2PD	Convert packed single precision floating-point values to packed double precision floating-point values.
CVTPD2PS	Convert packed double precision floating-point values to packed single precision floating-point values.
CVTSS2SD	Convert scalar single precision floating-point values to scalar double precision floating-point values.

CVTSD2SS	Convert scalar double precision floating-point values to scalar single precision floating-point values.
CVTSD2SI	Convert scalar double precision floating-point values to a signed integer.
CVTTSD2SI	Convert with truncation scalar double precision floating-point values to a scalar signed integer.
CVTSI2SD	Convert signed integer to scalar double precision floating-point value.

5.6.2 Intel® SSE2 Packed Single Precision Floating-Point Instructions

Intel SSE2 packed single precision floating-point instructions perform conversion operations on single precision floating-point and integer operands. These instructions represent enhancements to the Intel SSE single precision floating-point instructions.

CVTDQ2PS	Convert packed doubleword integers to packed single precision floating-point values.
CVTPS2DQ	Convert packed single precision floating-point values to packed doubleword integers.
CVTTPS2DQ	Convert with truncation packed single precision floating-point values to packed doubleword integers.

5.6.3 Intel® SSE2 128-Bit SIMD Integer Instructions

Intel SSE2 SIMD integer instructions perform additional operations on packed words, doublewords, and quadwords contained in XMM and MMX registers.

MOVDQA	Move aligned double quadword.
MOVDQU	Move unaligned double quadword.
MOVQ2DQ	Move quadword integer from MMX to XMM registers.
MOVDQ2Q	Move quadword integer from XMM to MMX registers.
PMULUDQ	Multiply packed unsigned doubleword integers.
PADDQ	Add packed quadword integers.
PSUBQ	Subtract packed quadword integers.
PSHUFLW	Shuffle packed low words.
PSHUFHW	Shuffle packed high words.
PSHUFD	Shuffle packed doublewords.
PSLLDQ	Shift double quadword left logical.
PSRLDQ	Shift double quadword right logical.
PUNPCKHQDQ	Unpack high quadwords.
PUNPCKLQDQ	Unpack low quadwords.

5.6.4 Intel® SSE2 Cacheability Control and Ordering Instructions

Intel SSE2 cacheability control instructions provide additional operations for caching of non-temporal data when storing data from XMM registers to memory. LFENCE and MFENCE provide additional control of instruction ordering on store operations.

CLFLUSH	See Section 5.1.13.
LFENCE	Serializes load operations.
MFENCE	Serializes load and store operations.
PAUSE	Improves the performance of “spin-wait loops”.
MASKMOVDQU	Non-temporal store of selected bytes from an XMM register into memory.
MOVNTPD	Non-temporal store of two packed double precision floating-point values from an XMM register into memory.
MOVNTDQ	Non-temporal store of double quadword from an XMM register into memory.

MOVNTI	Non-temporal store of a doubleword from a general-purpose register into memory.

5.7 INTEL® SSE3 INSTRUCTIONS

The Intel SSE3 extensions offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities. These instructions can be grouped into the following categories:

- One x87 FPU instruction used in integer conversion.
- One SIMD integer instruction that addresses unaligned data loads.
- Two SIMD floating-point packed ADD/SUB instructions.
- Four SIMD floating-point horizontal ADD/SUB instructions.
- Three SIMD floating-point LOAD/MOVE/DUPLICATE instructions.
- Two thread synchronization instructions.

Intel SSE3 instructions can only be executed on Intel 64 and IA-32 processors that support Intel SSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

The sections that follow describe each subgroup.

5.7.1 Intel® SSE3 x87-FP Integer Conversion Instruction

FISTTP	Behaves like the FISTP instruction but uses truncation, irrespective of the rounding mode specified in the floating-point control word (FCW).

5.7.2 Intel® SSE3 Specialized 128-Bit Unaligned Data Load Instruction

LDDQU	Special 128-bit unaligned load designed to avoid cache line splits.

5.7.3 Intel® SSE3 SIMD Floating-Point Packed ADD/SUB Instructions

ADDSUBPS	Performs single precision addition on the second and fourth pairs of 32-bit data elements within the operands; single precision subtraction on the first and third pairs.
ADDSUBPD	Performs double precision addition on the second pair of quadwords, and double precision subtraction on the first pair.

5.7.4 Intel® SSE3 SIMD Floating-Point Horizontal ADD/SUB Instructions

HADDPS	Performs a single precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.
HSUBPS	Performs a single precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by subtracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand.

HADDPD	Performs a double precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand.
HSUBPD	Performs a double precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand.

5.7.5 Intel® SSE3 SIMD Floating-Point LOAD/MOVE/DUPLICATE Instructions

MOVSHDUP	Loads/moves 128 bits; duplicating the second and fourth 32-bit data elements.
MOVSLDUP	Loads/moves 128 bits; duplicating the first and third 32-bit data elements.
MOVDDUP	Loads/moves 64 bits (bits[63:0] if the source is a register) and returns the same 64 bits in both the lower and upper halves of the 128-bit result register; duplicates the 64 bits from the source.

5.7.6 Intel® SSE3 Agent Synchronization Instructions

MONITOR	Sets up an address range used to monitor write-back stores.
MWAIT	Enables a logical processor to enter into an optimized state while waiting for a write-back store to the address range set up by the MONITOR instruction.

5.8 SUPPLEMENTAL STREAMING SIMD EXTENSIONS 3 (SSSE3) INSTRUCTIONS

SSSE3 provide 32 instructions (represented by 14 mnemonics) to accelerate computations on packed integers. These include:

- Twelve instructions that perform horizontal addition or subtraction operations.
- Six instructions that evaluate absolute values.
- Two instructions that perform multiply and add operations and speed up the evaluation of dot products.
- Two instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- Two instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- Six instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- Two instructions that align data from the composite of two operands.

SSSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

The sections that follow describe each subgroup.

5.8.1 Horizontal Addition/Subtraction

PHADDW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.
PHADDSW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.
PHADDD	Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.
PHSUBW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the

	source and destination operands. The signed 16-bit results are packed and written to the destination operand.
PHSUBSW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand.
PHSUBD	Performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

5.8.2 Packed Absolute Values

PABSB	Computes the absolute value of each signed byte data element.
PABSW	Computes the absolute value of each signed 16-bit data element.
PABSD	Computes the absolute value of each signed 32-bit data element.

5.8.3 Multiply and Add Packed Signed and Unsigned Bytes

PMADDUBSW	Multiplies each unsigned byte value with the corresponding signed byte value to produce an intermediate, 16-bit signed integer. Each adjacent pair of 16-bit signed values are added horizontally. The signed, saturated 16-bit results are packed to the destination operand.

5.8.4 Packed Multiply High with Round and Scale

PMULHRSW	Multiplies vertically each signed 16-bit integer from the destination operand with the corresponding signed 16-bit integer of the source operand, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

5.8.5 Packed Shuffle Bytes

PSHUFB	Permutes each byte in place, according to a shuffle control mask. The least significant three or four bits of each shuffle control byte of the control mask form the shuffle index. The shuffle mask is unaffected. If the most significant bit (bit 7) of a shuffle control byte is set, the constant zero is written in the result byte.

5.8.6 Packed Sign

PSIGNB/W/D	Negates each signed integer element of the destination operand if the sign of the corresponding data element in the source operand is less than zero.

5.8.7 Packed Align Right

PALIGNR	Source operand is appended after the destination operand forming an intermediate value of twice the width of an operand. The result is extracted from the intermediate value into the destination operand by selecting the 128-bit or 64-bit value that are right-aligned to the byte offset specified by the immediate value.

5.9 INTEL® SSE4 INSTRUCTIONS

Intel Streaming SIMD Extensions 4 (Intel SSE4) introduces 54 new instructions. 47 of the Intel SSE4 instructions are referred to as Intel SSE4.1 in this document, and 7 new Intel SSE4 instructions are referred to as Intel SSE4.2.

Intel SSE4.1 is targeted to improve the performance of media, imaging, and 3D workloads. Intel SSE4.1 adds instructions that improve compiler vectorization and significantly increase support for packed dword computation. The technology also provides a hint that can improve memory throughput when reading from uncacheable WC memory type.

The 47 Intel SSE4.1 instructions include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction performs a load with a streaming hint.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

The Intel SSE4.2 instructions operating on XMM registers include:

- String and text processing that can take advantage of single-instruction multiple-data programming techniques.
- A SIMD integer instruction that enhances the capability of the 128-bit integer SIMD capability in SSE4.1.

5.10 INTEL® SSE4.1 INSTRUCTIONS

Intel SSE4.1 instructions can use an XMM register as a source or destination. Programming Intel SSE4.1 is similar to programming 128-bit Integer SIMD and floating-point SIMD instructions in Intel SSE/SSE2/SSE3/SSSE3. Intel SSE4.1 does not provide any 64-bit integer SIMD instructions operating on MMX registers. The sections that follow describe each subgroup.

5.10.1 Dword Multiply Instructions

PMULLD	Returns four lower 32-bits of the 64-bit results of signed 32-bit integer multiplies.
PMULDQ	Returns two 64-bit signed result of signed 32-bit integer multiplies.

5.10.2 Floating-Point Dot Product Instructions

DPPD	Perform double precision dot product for up to 2 elements and broadcast.
DPPS	Perform single precision dot products for up to 4 elements and broadcast.

5.10.3 Streaming Load Hint Instruction

MOVNTDQA	Provides a non-temporal hint that can cause adjacent 16-byte items within an aligned 64-byte region (a streaming line) to be fetched and held in a small set of temporary buffers

("streaming load buffers"). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be supplied from the streaming load buffer and can improve throughput.

5.10.4 Packed Blending Instructions

BLENDPD	Conditionally copies specified double precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDPS	Conditionally copies specified single precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDVPD	Conditionally copies specified double precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
BLENDVPS	Conditionally copies specified single precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
PBLENDVB	Conditionally copies specified byte elements in the source operand to the corresponding elements in the destination, using an implied mask.
PBLENDW	Conditionally copies specified word elements in the source operand to the corresponding elements in the destination, using an immediate byte control.

5.10.5 Packed Integer MIN/MAX Instructions

PMINUW	Compare packed unsigned word integers.
PMINUD	Compare packed unsigned dword integers.
PMINSB	Compare packed signed byte integers.
PMINSD	Compare packed signed dword integers.
PMAXUW	Compare packed unsigned word integers.
PMAXUD	Compare packed unsigned dword integers.
PMAXSB	Compare packed signed byte integers.
PMAXSD	Compare packed signed dword integers.

5.10.6 Floating-Point Round Instructions With Selectable Rounding Mode

ROUNDPS	Round packed single precision floating-point values into integer values and return rounded floating-point values.
ROUNDPD	Round packed double precision floating-point values into integer values and return rounded floating-point values.
ROUNDSS	Round the low packed single precision floating-point value into an integer value and return a rounded floating-point value.
ROUNDSD	Round the low packed double precision floating-point value into an integer value and return a rounded floating-point value.

5.10.7 Insertion and Extractions from XMM Registers

EXTRACTPS	Extracts a single precision floating-point value from a specified offset in an XMM register and stores the result to memory or a general-purpose register.
INSERTPS	Inserts a single precision floating-point value from either a 32-bit memory location or selected from a specified offset in an XMM register to a specified offset in the destination XMM register. In addition, INSERTPS allows zeroing out selected data elements in the destination, using a mask.

PINSRB	Insert a byte value from a register or memory into an XMM register.
PINSRD	Insert a dword value from 32-bit register or memory into an XMM register.
PINSRQ	Insert a qword value from 64-bit register or memory into an XMM register.
PEXTRB	Extract a byte from an XMM register and insert the value into a general-purpose register or memory.
PEXTRW	Extract a word from an XMM register and insert the value into a general-purpose register or memory.
PEXTRD	Extract a dword from an XMM register and insert the value into a general-purpose register or memory.
PEXTRQ	Extract a qword from an XMM register and insert the value into a general-purpose register or memory.

5.10.8 Packed Integer Format Conversions

PMOVSXBW	Sign extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVZXBW	Zero extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVSXBD	Sign extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVZXBD	Zero extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVSXWD	Sign extend the lower 16-bit integer of each packed dword element into packed signed dword integers.
PMOVZXWD	Zero extend the lower 16-bit integer of each packed dword element into packed signed dword integers.
PMOVSXBQ	Sign extend the lower 8-bit integer of each packed qword element into packed signed qword integers.
PMOVZXBQ	Zero extend the lower 8-bit integer of each packed qword element into packed signed qword integers.
PMOVSXWQ	Sign extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVZXWQ	Zero extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVSXDQ	Sign extend the lower 32-bit integer of each packed qword element into packed signed qword integers.
PMOVZXDQ	Zero extend the lower 32-bit integer of each packed qword element into packed signed qword integers.

5.10.9 Improved Sums of Absolute Differences (SAD) for 4-Byte Blocks

MPSADBW	Performs eight 4-byte wide Sum of Absolute Differences operations to produce eight word integers.

5.10.10 Horizontal Search

PHMINPOSUW	Finds the value and location of the minimum unsigned word from one of 8 horizontally packed unsigned words. The resulting value and location (offset within the source) are packed into the low dword of the destination XMM register.

5.10.11 Packed Test

PTEST	Performs a logical AND between the destination with this mask and sets the ZF flag if the result is zero. The CF flag (zero for TEST) is set if the inverted mask AND'd with the destination is all zeroes.

5.10.12 Packed Qword Equality Comparisons

PCMPEQQ	128-bit packed qword equality test.

5.10.13 Dword Packing With Unsigned Saturation

PACKUSDW	Packs dword to word with unsigned saturation.

5.11 INTEL® SSE4.2 INSTRUCTION SET

Five of the Intel SSE4.2 instructions operate on XMM register as a source or destination. These include four text/string processing instructions and one packed quadword compare SIMD instruction. Programming these five Intel SSE4.2 instructions is similar to programming 128-bit Integer SIMD in Intel SSE2/SSSE3. Intel SSE4.2 does not provide any 64-bit integer SIMD instructions.

CRC32 operates on general-purpose registers and is summarized in Section 5.1.6. The sections that follow summarize each subgroup.

5.11.1 String and Text Processing Instructions

PCMPESTRI	Packed compare explicit-length strings, return index in ECX/RCX.
PCMPESTRM	Packed compare explicit-length strings, return mask in XMM0.
PCMPISTRI	Packed compare implicit-length strings, return index in ECX/RCX.
PCMPISTRM	Packed compare implicit-length strings, return mask in XMM0.

5.11.2 Packed Comparison SIMD Integer Instruction

PCMPGTQ	Performs logical compare of greater-than on packed integer quadwords.

5.12 INTEL® AES-NI AND PCLMULQDQ

Six Intel® AES-NI instructions operate on XMM registers to provide accelerated primitives for block encryption/decryption using Advanced Encryption Standard (FIPS-197). The PCLMULQDQ instruction performs carry-less multiplication for two binary numbers up to 64-bit wide.

AESDEC	Perform an AES decryption round using an 128-bit state and a round key.
AESDECLAST	Perform the last AES decryption round using an 128-bit state and a round key.
AESENC	Perform an AES encryption round using an 128-bit state and a round key.
AESENCLAST	Perform the last AES encryption round using an 128-bit state and a round key.
AESIMC	Perform an inverse mix column transformation primitive.
AESKEYGENASSIST	Assist the creation of round keys with a key expansion schedule.
PCLMULQDQ	Perform carryless multiplication of two 64-bit numbers.

5.13 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel® Advanced Vector Extensions (AVX) promote legacy 128-bit SIMD instruction sets that operate on the XMM register set to use a "vector extension" (VEX) prefix and operates on 256-bit vector registers (YMM). Almost all prior generations of 128-bit SIMD instructions that operate on XMM (but not on MMX registers) are promoted to support three-operand syntax with VEX-128 encoding.

VEX-prefix encoded Intel AVX instructions support 256-bit and 128-bit floating-point operations by extending the legacy 128-bit SIMD floating-point instructions to support three-operand syntax.

Additional functional enhancements are also provided with VEX-encoded Intel AVX instructions.

The list of Intel AVX instructions is included in the following tables:

- Table 14-2 lists 256-bit and 128-bit floating-point arithmetic instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-3 lists 256-bit and 128-bit data movement and processing instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-4 lists functional enhancements of 256-bit Intel AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-5 lists 128-bit integer and floating-point instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-6 lists functional enhancements of 128-bit Intel AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-7 lists 128-bit data movement and processing instructions promoted from legacy instruction sets.

5.14 16-BIT FLOATING-POINT CONVERSION

Conversions between single precision floating-point (32-bit) and half precision floating-point (16-bit) data are provided by the VCVTPS2PH and VCVTPH2PS instructions, introduced beginning with the third generation of Intel Core processors based on Ivy Bridge microarchitecture:

VCVTPH2PS	Convert eight/four data elements containing 16-bit floating-point data into eight/four single precision floating-point data.
VCVTPS2PH	Convert eight/four data elements containing single precision floating-point data into eight/four 16-bit floating-point data.

Starting with the 4th generation Intel Xeon Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® AVX-512 instruction set architecture for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16). Section 5.19 includes a list of these instructions.

5.15 FUSED-MULTIPLY-ADD (FMA)

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

- Table 14-15 lists FMA instruction sets.

5.16 INTEL® ADVANCED VECTOR EXTENSIONS 2 (INTEL® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

- Table 14-18 lists promoted vector integer instructions in AVX2.
- Table 14-19 lists new instructions in AVX2 that complements AVX.

5.17 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX)

XABORT	Abort an RTM transaction execution.
XACQUIRE	Prefix hint to the beginning of an HLE transaction region.
XRELEASE	Prefix hint to the end of an HLE transaction region.
XBEGIN	Transaction begin of an RTM transaction region.
XEND	Transaction end of an RTM transaction region.
XTEST	Test if executing in a transactional region.
XRESLDTRK	Resume tracking load addresses.
XSUSLDTRK	Suspend tracking load addresses.

5.18 INTEL® SHA EXTENSIONS

Intel® SHA extensions provide a set of instructions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants.

SHA1MSG1	Perform an intermediate calculation for the next four SHA1 message dwords from the previous message dwords.
SHA1MSG2	Perform the final calculation for the next four SHA1 message dwords from the intermediate message dwords.
SHA1NEXTE	Calculate SHA1 state E after four rounds.
SHA1RNDS4	Perform four rounds of SHA1 operations.
SHA256MSG1	Perform an intermediate calculation for the next four SHA256 message dwords.
SHA256MSG2	Perform the final calculation for the next four SHA256 message dwords.
SHA256RNDS2	Perform two rounds of SHA256 operations.

5.19 INTEL® ADVANCED VECTOR EXTENSIONS 512 (INTEL® AVX-512)

The Intel® AVX-512 family comprises a collection of 512-bit SIMD instruction sets to accelerate a diverse range of applications. Intel AVX-512 instructions provide a wide range of functionality that support programming in 512-bit, 256 and 128-bit vector register, plus support for opmask registers and instructions operating on opmask registers.

The collection of 512-bit SIMD instruction sets in Intel AVX-512 include new functionality not available in Intel AVX and Intel AVX2, and promoted instructions similar to equivalent ones in Intel AVX/Intel AVX2 but with enhancement provided by opmask registers not available to VEX-encoded Intel AVX/Intel AVX2. Some instruction mnemonics in Intel AVX/Intel AVX2 that are promoted into Intel AVX-512 can be replaced by new instruction mnemonics that are available only with EVEX encoding, e.g., VBROADCASTF128 into VBROADCASTF32X4. Details of EVEX instruction encoding are discussed in Section 2.7, "Intel® AVX-512 Encoding," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. Starting with the 4th generation Intel Xeon Scalable Processor Family, an Intel AVX-512 instruction set architecture for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values, which complements the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products.

512-bit instruction mnemonics in AVX-512F instructions that are not Intel AVX or AVX2 promotions include:

VALIGND/Q	Perform dword/qword alignment of two concatenated source vectors.
VBLENDMPD/PS	Replace the VBLENDVPD/PS instructions (using opmask as select control).

VCOMPRESSPD/PS	Compress packed DP or SP elements of a vector.
VCVT(T)PD2UDQ	Convert packed DP FP elements of a vector to packed unsigned 32-bit integers.
VCVT(T)PS2UDQ	Convert packed SP FP elements of a vector to packed unsigned 32-bit integers.
VCVTQQ2PD/PS	Convert packed signed 64-bit integers to packed DP/SP FP elements.
VCVT(T)SD2USI	Convert the low DP FP element of a vector to an unsigned integer.
VCVT(T)SS2USI	Convert the low SP FP element of a vector to an unsigned integer.
VCVTUDQ2PD/PS	Convert packed unsigned 32-bit integers to packed DP/SP FP elements.
VCVTUSI2USD/S	Convert an unsigned integer to the low DP/SP FP element and merge to a vector.
VEXPANDPD/PS	Expand packed DP or SP elements of a vector.
VEXTRACTF32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VEXTRACTI32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VFIXUPIMMPD/PS	Perform fix-up to special values in DP/SP FP vectors.
VFIXUPIMMSD/SS	Perform fix-up to special values of the low DP/SP FP element.
VGETEXPPD/PS	Convert the exponent of DP/SP FP elements of a vector into FP values.
VGETEXPSD/SS	Convert the exponent of the low DP/SP FP element in a vector into FP value.
VGETMANTPD/PS	Convert the mantissa of DP/SP FP elements of a vector into FP values.
VGETMANTSD/SS	Convert the mantissa of the low DP/SP FP element of a vector into FP value.
VINSERTF32X4/64X4	Insert a 128/256-bit vector into a full-length vector with 32/64-bit granular update.
VMOVDQA32/64	VMOVDQA with 32/64-bit granular conditional update.
VMOVDQU32/64	VMOVDQU with 32/64-bit granular conditional update.
VPBLENDMD/Q	Blend dword/qword elements using opmask as select control.
VPBROADCASTD/Q	Broadcast from general-purpose register to vector register.
VPCMPD/UD	Compare packed signed/unsigned dwords using specified primitive.
VPCMPQ/UQ	Compare packed signed/unsigned quadwords using specified primitive.
VPCOMPRESSQ/D	Compress packed 64/32-bit elements of a vector.
VPERMI2D/Q	Full permute of two tables of dword/qword elements overwriting the index vector.
VPERMI2PD/PS	Full permute of two tables of DP/SP elements overwriting the index vector.
VPERMT2D/Q	Full permute of two tables of dword/qword elements overwriting one source table.
VPERMT2PD/PS	Full permute of two tables of DP/SP elements overwriting one source table.
VPEXPANDD/Q	Expand packed dword/qword elements of a vector.
VPMAXSQ	Compute maximum of packed signed 64-bit integer elements.
VPMAXUD/UQ	Compute maximum of packed unsigned 32/64-bit integer elements.
VPMINSQ	Compute minimum of packed signed 64-bit integer elements.
VPMINUD/UQ	Compute minimum of packed unsigned 32/64-bit integer elements.
VPMOV(S|US)QB	Down convert qword elements in a vector to byte elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)QW	Down convert qword elements in a vector to word elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)QD	Down convert qword elements in a vector to dword elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)DB	Down convert dword elements in a vector to byte elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)DW	Down convert dword elements in a vector to word elements using truncation (saturation | unsigned saturation).
VPROLD/Q	Rotate dword/qword element left by a constant shift count with conditional update.
VPROLVD/Q	Rotate dword/qword element left by shift counts specified in a vector with conditional update.
VPRORD/Q	Rotate dword/qword element right by a constant shift count with conditional update.

VPRORRD/Q	Rotate dword/qword element right by shift counts specified in a vector with conditional update.
VPSCATTERDD/DQ	Scatter dword/qword elements in a vector to memory using dword indices.
VPSCATTERQD/QQ	Scatter dword/qword elements in a vector to memory using qword indices.
VPSRAQ	Shift qwords right by a constant shift count and shifting in sign bits.
VPSRAVQ	Shift qwords right by shift counts in a vector and shifting in sign bits.
VPTESTNMD/Q	Perform bitwise NAND of dword/qword elements of two vectors and write results to opmask.
VPTERLOGD/Q	Perform bitwise ternary logic operation of three vectors with 32/64 bit granular conditional update.
VPTESTMD/Q	Perform bitwise AND of dword/qword elements of two vectors and write results to opmask.
VRCP14PD/PS	Compute approximate reciprocals of packed DP/SP FP elements of a vector.
VRCP14SD/SS	Compute the approximate reciprocal of the low DP/SP FP element of a vector.
VRNDSCALEPD/PS	Round packed DP/SP FP elements of a vector to specified number of fraction bits.
VRNDSCALESD/SS	Round the low DP/SP FP element of a vector to specified number of fraction bits.
VRSQRT14PD/PS	Compute approximate reciprocals of square roots of packed DP/SP FP elements of a vector.
VRSQRT14SD/SS	Compute the approximate reciprocal of square root of the low DP/SP FP element of a vector.
VSCALEPD/PS	Multiply packed DP/SP FP elements of a vector by powers of two with exponents specified in a second vector.
VSCALESD/SS	Multiply the low DP/SP FP element of a vector by powers of two with exponent specified in the corresponding element of a second vector.
VSCATTERDD/DQ	Scatter SP/DP FP elements in a vector to memory using dword indices.
VSCATTERQD/QQ	Scatter SP/DP FP elements in a vector to memory using qword indices.
VSHUFF32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.
VSHUFI32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.

512-bit instruction mnemonics in AVX-512DQ that are not Intel AVX or AVX2 promotions include:

VCVT(T)PD2QQ	Convert packed DP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PD2UQQ	Convert packed DP FP elements of a vector to packed unsigned 64-bit integers.
VCVT(T)PS2QQ	Convert packed SP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PS2UQQ	Convert packed SP FP elements of a vector to packed unsigned 64-bit integers.
VCVTUQQ2PD/PS	Convert packed unsigned 64-bit integers to packed DP/SP FP elements.
VEXTRACTF64X2	Extract a vector from a full-length vector with 64-bit granular update.
VEXTRACTI64X2	Extract a vector from a full-length vector with 64-bit granular update.
VFPCLASSPD/PS	Test packed DP/SP FP elements in a vector by numeric/special-value category.
VFPCLASSSD/SS	Test the low DP/SP FP element by numeric/special-value category.
VINSERTF64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VINSERTI64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VPMOVM2D/Q	Convert opmask register to vector register in 32/64-bit granularity.
VPMOVB2D/Q2M	Convert a vector register in 32/64-bit granularity to an opmask register.
VPMULLQ	Multiply packed signed 64-bit integer elements of two vectors and store low 64-bit signed result.
VRANGEPD/PS	Perform RANGE operation on each pair of DP/SP FP elements of two vectors using specified range primitive in imm8.
VRANGESD/SS	Perform RANGE operation on the pair of low DP/SP FP element of two vectors using specified range primitive in imm8.

VREDUCEPD/PS	Perform Reduction operation on packed DP/SP FP elements of a vector using specified reduction primitive in imm8.
VREDUCESD/SS	Perform Reduction operation on the low DP/SP FP element of a vector using specified reduction primitive in imm8.

512-bit instruction mnemonics in AVX-512BW that are not Intel AVX or AVX2 promotions include:

VDBPSADBW	Double block packed Sum-Absolute-Differences on unsigned bytes.
VMOVDQU8/16	VMOVDQU with 8/16-bit granular conditional update.
VPBLENDMB	Replaces the VPBLENDVB instruction (using opmask as select control).
VPBLENDMW	Blend word elements using opmask as select control.
VPBROADCASTB/W	Broadcast from general-purpose register to vector register.
VPCMPB/UB	Compare packed signed/unsigned bytes using specified primitive.
VPCMPW/UW	Compare packed signed/unsigned words using specified primitive.
VPERMW	Permute packed word elements.
VPERMI2B/W	Full permute from two tables of byte/word elements overwriting the index vector.
VPMOVM2B/W	Convert opmask register to vector register in 8/16-bit granularity.
VPMOVB2M/W2M	Convert a vector register in 8/16-bit granularity to an opmask register.
VPMOV(S|US)WB	Down convert word elements in a vector to byte elements using truncation (saturation | unsigned saturation).
VPSLLVW	Shift word elements in a vector left by shift counts in a vector.
VPSRAVW	Shift words right by shift counts in a vector and shifting in sign bits.
VPSRLVW	Shift word elements in a vector right by shift counts in a vector.
VPTESTNMB/W	Perform bitwise NAND of byte/word elements of two vectors and write results to opmask.
VPTESTMB/W	Perform bitwise AND of byte/word elements of two vectors and write results to opmask.

512-bit instruction mnemonics in AVX-512CD that are not Intel AVX or AVX2 promotions include:

VPBROADCASTM	Broadcast from opmask register to vector register.
VPCONFLICTD/Q	Detect conflicts within a vector of packed 32/64-bit integers.
VPLZCNTD/Q	Count the number of leading zero bits of packed dword/qword elements.

Opmask instructions include:

KADDB/W/D/Q	Add two 8/16/32/64-bit opmasks.
KANDB/W/D/Q	Logical AND two 8/16/32/64-bit opmasks.
KANDNB/W/D/Q	Logical AND NOT two 8/16/32/64-bit opmasks.
KMOVB/W/D/Q	Move from or move to opmask register of 8/16/32/64-bit data.
KNOTB/W/D/Q	Bitwise NOT of two 8/16/32/64-bit opmasks.
KORB/W/D/Q	Logical OR two 8/16/32/64-bit opmasks.
KORTESTB/W/D/Q	Update EFLAGS according to the result of bitwise OR of two 8/16/32/64-bit opmasks.
KSHIFTLB/W/D/Q	Shift left 8/16/32/64-bit opmask by specified count.
KSHIFTRB/W/D/Q	Shift right 8/16/32/64-bit opmask by specified count.
KTESTB/W/D/Q	Update EFLAGS according to the result of bitwise TEST of two 8/16/32/64-bit opmasks.
KUNPCKBW/WD/DQ	Unpack and interleave two 8/16/32-bit opmasks into 16/32/64-bit mask.
KXNORB/W/D/Q	Bitwise logical XNOR of two 8/16/32/64-bit opmasks.
KXORB/W/D/Q	Logical XOR of two 8/16/32/64-bit opmasks.

512-bit instruction mnemonics in AVX-512ER include:

VEXP2PD/PS	Compute approximate base-2 exponential of packed DP/SP FP elements of a vector.
VEXP2SD/SS	Compute approximate base-2 exponential of the low DP/SP FP element of a vector.
VRCP28PD/PS	Compute approximate reciprocals to 28 bits of packed DP/SP FP elements of a vector.
VRCP28SD/SS	Compute the approximate reciprocal to 28 bits of the low DP/SP FP element of a vector.
VRSQRT28PD/PS	Compute approximate reciprocals of square roots to 28 bits of packed DP/SP FP elements of a vector.
VRSQRT28SD/SS	Compute the approximate reciprocal of square root to 28 bits of the low DP/SP FP element of a vector.

512-bit instruction mnemonics in AVX-512PF include:

VGATHERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using dword indices.
VGATHERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using qword indices.
VGATHERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using dword indices.
VGATHERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using qword indices.
VSCATTERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using dword indices.
VSCATTERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using qword indices.
VSCATTERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using dword indices.
VSCATTERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using qword indices.

512-bit instruction mnemonics in AVX512-FP16 include:

VADDPH/SH	Add packed/scalar FP16 values.
VCMPPH/SH	Compare packed/scalar FP16 values.
VCOMISH	Compare scalar ordered FP16 values and set EFLAGS.
VCVTDQ2PH	Convert packed signed doubleword integers to packed FP16 values.
VCVTPD2PH	Convert packed double precision FP values to packed FP16 values.
VCVTPH2DQ/QQ	Convert packed FP16 values to signed doubleword/quadword integers.
VCVTPH2PD	Convert packed FP16 values to FP64 values.
VCVTPH2PS[X]	Convert packed FP16 values to single precision floating-point values.
VCVTPH2QQ	Convert packed FP16 values to signed quadword integer values.
VCVTPH2UDQ/QQ	Convert packed FP16 values to unsigned doubleword/quadword integers.
VCVTPH2UW/W	Convert packed FP16 values to unsigned/signed word integers.
VCVTPS2PH[X]	Convert packed single precision floating-point values to packed FP16 values.
VCVTQQ2PH	Convert packed signed quadword integers to packed FP16 values.
VCVTSD2SH	Convert low FP64 value to an FP16 value.
VCVTSH2SD/SS	Convert low FP16 value to an FP64/FP32 value.
VCVTSH2SI/USI	Convert low FP16 value to signed/unsigned integer.
VCVTSI2SH	Convert a signed doubleword/quadword integer to an FP16 value.
VCVTSS2SH	Convert low FP32 value to an FP16 value.
VCVTTPH2DQ/QQ	Convert with truncation packed FP16 values to signed doubleword/quadword integers.
VCVTTPH2UDQ/QQ	Convert with truncation packed FP16 values to unsigned doubleword/quadword integers.
VCVTTPH2UW/W	Convert packed FP16 values to unsigned/signed word integers.
VCVTTSH2SI/USI	Convert with truncation low FP16 value to a signed/unsigned integer.
VCVTUDQ2PH	Convert packed unsigned doubleword integers to packed FP16 values.
VCVTUQQ2PH	Convert packed unsigned quadword integers to packed FP16 values.
VCVTUSI2SH	Convert unsigned doubleword integer to an FP16 value.
VCVTUW2PH	Convert packed unsigned word integers to FP16 values.
VCVTW2PH	Convert packed signed word integers to FP16 values.

VDIVPH/SH	Divide packed/scalar FP16 values.
VF[C]MADDCPH	Complex multiply and accumulate FP16 values.
VF[C]MADDCSH	Complex multiply and accumulate scalar FP16 values.
VF[C]MULCPH	Complex multiply FP16 values.
VF[C]MULCSH	Complex multiply scalar FP16 values.
VF[,N]MADD[132,213,231]PH	Fused multiply-add of packed FP16 values.
VF[,N]MADD[132,213,231]SH	Fused multiply-add of scalar FP16 values.
VFMADDSUB[132,213,231]PH	Fused multiply-alternating add/subtract of packed FP16 values.
VFMSUBADD[132,213,231]PH	Fused multiply-alternating subtract/add of packed FP16 values.
VF[,N]MSUB[132,213,231]PH	Fused multiply-subtract of packed FP16 values.
VF[,N]MSUB[132,213,231]SH	Fused multiply-subtract of scalar FP16 values.
VFPCLASSPH/SH	Test types of packed/scalar FP16 values.
VGETEXPPH/SH	Convert exponents of packed/scalar FP16 values to FP16 values.
VGETMANTPH/SH	Extract FP16 vector of normalized mantissas from FP16 vector/scalar.
VMAXPH/PS	Return maximum of packed/scalar FP16 values.
VMINPH/PS	Return minimum of packed/scalar FP16 values.
VMOVSH	Move scalar FP16 value.
VMOVW	Move word.
VMULPH/SH	Multiply packed/scalar FP16 values.
VRCPPH/SH	Compute reciprocals of packed/scalar FP16 values.
VREDUCEPH/SH	Perform reduction transformation on packed/scalar FP16 values.
VRNDSCALEPH/SH	Round packed/scalar FP16 values to include a given number of fraction bits.
VRSQRTPH/SH	Compute reciprocals of square roots of packed/scalar FP16 values.
VSCALEPH/SH	Scale packed/scalar FP16 values with FP16 values.
VSQRTPH/SH	Compute square root of packed/scalar FP16 values.
VSUBPH/SH	Subtract packed/scalar FP16 values.
VUCOMISH	Unordered compare scalar FP16 values and set EFLAGS.

5.20 SYSTEM INSTRUCTIONS

The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

CLAC	Clear AC Flag in EFLAGS register.
STAC	Set AC Flag in EFLAGS register.
LGDT	Load global descriptor table (GDT) register.
SGDT	Store global descriptor table (GDT) register.
LLDT	Load local descriptor table (LDT) register.
SLDT	Store local descriptor table (LDT) register.
LTR	Load task register.
STR	Store task register.
LIDT	Load interrupt descriptor table (IDT) register.
SIDT	Store interrupt descriptor table (IDT) register.
MOV	Load and store control registers.
LMSW	Load machine status word.
SMSW	Store machine status word.
CLTS	Clear the task-switched flag.

ARPL	Adjust requested privilege level.
LAR	Load access rights.
LSL	Load segment limit.
VERR	Verify segment for reading
VERW	Verify segment for writing.
MOV	Load and store debug registers.
INVD	Invalidate cache, no writeback.
WBINVD	Invalidate cache, with writeback.
INVLPG	Invalidate TLB Entry.
INVPCID	Invalidate Process-Context Identifier.
LOCK (prefix)	Perform atomic access to memory (can be applied to a number of general purpose instructions that provide memory source/destination access).
HLT	Halt processor.
RSM	Return from system management mode (SMM).
RDMSR	Read model-specific register.
WRMSR	Write model-specific register.
RDPMC	Read performance monitoring counters.
RDTSC	Read time stamp counter.
RDTSCP	Read time stamp counter and processor ID.
SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL = 0.
SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL = 3.
XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XSAVES	Save processor supervisor-mode extended states to memory.
XRSTOR	Restore processor extended states from memory.
XRSTORS	Restore processor supervisor-mode extended states from memory.
XGETBV	Reads the state of an extended control register.
XSETBV	Writes the state of an extended control register.
RDFSBASE	Reads from FS base address at any privilege level.
RDGSBASE	Reads from GS base address at any privilege level.
WRFSBASE	Writes to FS base address at any privilege level.
WRGSBASE	Writes to GS base address at any privilege level.

5.21 64-BIT MODE INSTRUCTIONS

The following instructions are introduced in 64-bit mode. This mode is a sub-mode of IA-32e mode.

CDQE	Convert doubleword to quadword.
CMPSQ	Compare string operands.
CMPXCHG16B	Compare RDX:RAX with m128.
LODSQ	Load qword at address (R)SI into RAX.
MOVSQ	Move qword from address (R)SI to (R)DI.
MOVZX (64-bits)	Move bytes/words to doublewords/quadwords, zero-extension.
STOSQ	Store RAX at address RDI.
SWAPGS	Exchanges current GS base register value with value in MSR address C0000102H.
SYSCALL	Fast call to privilege level 0 system procedures.

SYSRET	Return from fast system call.

5.22 VIRTUAL-MACHINE EXTENSIONS

The behavior of the VMCS-maintenance instructions is summarized below:

VMPTRLD	Takes a single 64-bit source operand in memory. It makes the referenced VMCS active and current.
VMPTRST	Takes a single 64-bit destination operand that is in memory. Current-VMCS pointer is stored into the destination operand.
VMCLEAR	Takes a single 64-bit operand in memory. The instruction sets the launch state of the VMCS referenced by the operand to "clear", renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region.
VMREAD	Reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand.
VMWRITE	Writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand.

The behavior of the VMX management instructions is summarized below:

VMLAUNCH	Launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMRESUME	Resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMXOFF	Causes the processor to leave VMX operation.
VMXON	Takes a single 64-bit source operand in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

The behavior of the VMX-specific TLB-management instructions is summarized below:

INVEPT	Invalidate cached **Extended Page Table** (EPT) mappings in the processor to synchronize address translation in virtual machines with memory-resident EPT pages.
INVVPID	Invalidate cached mappings of address translation based on the **Virtual Processor ID** (VPID).

None of the instructions above can be executed in compatibility mode; they generate invalid-opcode exceptions if executed in compatibility mode.

The behavior of the guest-available instructions is summarized below:

VMCALL	Allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
VMFUNC	Allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. No VM exit occurs.

5.23 SAFER MODE EXTENSIONS

The behavior of the GETSEC instruction leaves of the Safer Mode Extensions (SMX) are summarized below:

GETSEC[CAPABILITIES]	Returns the available leaf functions of the GETSEC instruction.
GETSEC[ENTERACCS]	Loads an authenticated code chipset module and enters authenticated code execution mode.
GETSEC[EXITAC]	Exits authenticated code execution mode.
GETSEC[SENTER]	Establishes a Measured Launched Environment (MLE) which has its dynamic root of trust anchored to a chipset supporting Intel Trusted Execution Technology.
GETSEC[SEXIT]	Exits the MLE.
GETSEC[PARAMETERS]	Returns SMX related parameter information.

GETSEC[SMCRTL]	SMX mode control.
GETSEC[WAKEUP]	Wakes up sleeping logical processors inside an MLE.

5.24 INTEL® MEMORY PROTECTION EXTENSIONS

Intel Memory Protection Extensions (Intel MPX) provides a set of instructions to enable software to add robust bounds checking capability to memory references. Details of Intel MPX are described in Appendix E, "Intel® Memory Protection Extensions."

BNDMK	Create a LowerBound and an UpperBound in a register.
BNDCL	Check the address of a memory reference against a LowerBound.
BNDCU	Check the address of a memory reference against an UpperBound in 1's complement form.
BNDCN	Check the address of a memory reference against an UpperBound not in 1's complement form.
BNDMOV	Copy or load from memory of the LowerBound and UpperBound to a register.
BNDMOV	Store to memory of the LowerBound and UpperBound from a register.
BNDLDX	Load bounds using address translation.
BNDSTX	Store bounds using address translation.

5.25 INTEL® SOFTWARE GUARD EXTENSIONS

Intel Software Guard Extensions (Intel SGX) provide two sets of instruction leaf functions to enable application software to instantiate a protected container, referred to as an enclave. The enclave instructions are organized as leaf functions under two instruction mnemonics: ENCLS (ring 0) and ENCLU (ring 3). Details of Intel SGX are described in Chapter 35 through Chapter 41 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D.

The first implementation of Intel SGX is also referred to as SGX1, it is introduced with the 6th Generation Intel Core Processors. The leaf functions supported in SGX1 are shown in Table 5-3.

Table 5-3. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add a page	ENCLU[EENTER]	Enter an Enclave
ENCLS[EBLOCK]	Block an EPC page	ENCLU[EEXIT]	Exit an Enclave
ENCLS[ECREATE]	Create an enclave	ENCLU[EGETKEY]	Create a cryptographic key
ENCLS[EDBGRD]	Read data by debugger	ENCLU[EREPORT]	Create a cryptographic report
ENCLS[EDBGWR]	Write data by debugger	ENCLU[ERESUME]	Re-enter an Enclave
ENCLS[EEXTEND]	Extend EPC page measurement		
ENCLS[EINIT]	Initialize an enclave		
ENCLS[ELDB]	Load an EPC page as blocked		
ENCLS[ELDU]	Load an EPC page as unblocked		
ENCLS[EPA]	Add version array		
ENCLS[EREMOVE]	Remove a page from EPC		
ENCLS[ETRACK]	Activate EBLOCK checks		
ENCLS[EWB]	Write back/invalidate an EPC page		

5.26 SHADOW STACK MANAGEMENT INSTRUCTIONS

Shadow stack management instructions allow the program and run-time to perform operations like recovering from control protection faults, shadow stack switching, etc. The following instructions are provided.

CLRSSBSY	Clear busy bit in a supervisor shadow stack token.
INCSSP	Increment the shadow stack pointer (SSP).
RDSSP	Read shadow stack point (SSP).
RSTORSSP	Restore a shadow stack pointer (SSP).
SAVEPREVSSP	Save previous shadow stack pointer (SSP).
SETSSBSY	Set busy bit in a supervisor shadow stack token.
WRSS	Write to a shadow stack.
WRUSS	Write to a user mode shadow stack.

5.27 CONTROL TRANSFER TERMINATING INSTRUCTIONS

ENDBR32	Terminate an Indirect Branch in 32-bit and Compatibility Mode.
ENDBR64	Terminate an Indirect Branch in 64-bit Mode.

5.28 INTEL® AMX INSTRUCTIONS

LDTILECFG	Load tile configuration.
STTILECFG	Store tile configuration.
TDPBF16PS	Dot product of BF16 tiles accumulated into packed single precision tile.
TDPBSSD	Dot product of signed bytes with dword accumulation.
TDPBSUD	Dot product of signed/unsigned bytes with dword accumulation.
TDPBUSD	Dot product of unsigned/signed bytes with dword accumulation.
TDPBUUD	Dot product of unsigned bytes with dword accumulation.
TILELOADD	Load data into tile.
TILELOADDT1	Load data into tile with hint to optimize data caching.
TILERELEASE	Release tile.
TILESTORED	Store tile.
TILEZERO	Zero tile.

5.29 USER INTERRUPT INSTRUCTIONS

CLUI	Clear user interrupt flag.
SENDUIPI	Send user interprocessor interrupt.
STUI	Set user interrupt flag.
TESTUI	Determine user interrupt flag.
UIRET	User-interrupt return.

5.30 ENQUEUE STORE INSTRUCTIONS

ENQCMD	Enqueue command.
ENQCMDS	Enqueue command supervisor.

5.31 INTEL® ADVANCED VECTOR EXTENSIONS 10 VERSION 1 INSTRUCTIONS

Intel® Advanced Vector Extensions 10 Version 1 (Intel® AVX10.1) is based on the Intel AVX-512 ISA feature set and includes all Intel AVX-512 instructions introduced with the Intel® Xeon® 6 P-core processor based on Granite Rapids microarchitecture. Intel AVX10.1 supports all instruction vector lengths (128, 256, and 512), as well as scalar and opmask instructions.

For a list of Intel AVX-512 instructions, see Section 5.19, “Intel® Advanced Vector Extensions 512 (Intel® AVX-512).” Additionally, note that some Intel AVX and Intel AVX2 instructions were promoted to Intel AVX512 and are also supported. See Section 5.13, “Intel® Advanced Vector Extensions (Intel® AVX),” Section 5.16, “Intel® Advanced Vector Extensions 2 (Intel® AVX2),” and Chapter 16, “Programming with Intel® AVX10,” for further details.

3. Updates to Chapter 16, Volume 1

Change bars and violet text show changes to Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

Changes to this chapter:

- Removed references to 256-bit maximum vector register size and enumeration of vector-length support.

CHAPTER 16 PROGRAMMING WITH INTEL® AVX10

16.1 INTRODUCTION

Intel® Advanced Vector Extensions 10 (Intel® AVX10) represents an enhancement to Intel® Advanced Vector Extensions 512 (Intel® AVX-512). Intel AVX10 establishes a common, converged vector instruction set across all Intel architectures, incorporating the modern vectorization aspects of Intel AVX-512.

Intel AVX10 is based on Intel AVX-512 and includes all Intel AVX-512 instructions. It supports all instruction vector lengths (128, 256, and 512), as well as scalar and opmask instructions.

16.2 FEATURE VERSIONING AND ENUMERATION

Most Intel AVX10 instructions and features will be organized in collections called **versions**. In some situations, a processor may introduce AVX10 instructions that are not part of that processor's AVX10 version. Such instructions will be enumerated discretely (see below).

AVX10 versions support enumeration that is monotonically increasing and inclusive. This can simplify application development by ensuring that all Intel processors support the same features and instructions at a given Intel AVX10 version number, as well as reduce the number of CPUID feature flags required to be checked by an application to determine feature support. In this enumeration paradigm, the application developer only need to check a CPUID feature flag indicating that the Intel AVX10 ISA is supported and a version number to ensure that the supported version is greater than or equal to the desired version.

The AVX10 feature flag indicates processor support for Intel AVX10 and the presence of a "Converged Vector ISA" leaf containing a field for the version number. AVX10 features or instructions that are not part of an AVX10 version when they are introduced will be enumerated with a discrete feature flag in that CPUID leaf. See Table 16-1 for details.

Table 16-1. CPUID Enumeration of Intel® AVX10

CPUID Bit	Description	Type
CPUID.(EAX=07H, ECX=01H):EDX[bit 19]	If 1, Intel® AVX10 is supported.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EAX[bits 31:0]	Reports the maximum supported sub-leaf.	Integer
CPUID.(EAX=24H, ECX=00H):EBX[bits 7:0]	Reports the Intel AVX10 version.	Integer (≥ 1)
CPUID.(EAX=24H, ECX=00H):EBX[bits 15:8]	Reserved.	N/A
CPUID.(EAX=24H, ECX=00H):EBX[bit 18:16]	Reserved.	Always 111B[1]
CPUID.(EAX=24H, ECX=00H):EBX[bits 31:19]	Reserved.	N/A
CPUID.(EAX=24H, ECX=00H):ECX[bits 31:0]	Reserved.	N/A
CPUID.(EAX=24H, ECX=00H):EDX[bits 31:0]	Reserved.	N/A
CPUID.(EAX=24H, ECX=01H):EAX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):EBX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):ECX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):EDX[bits 31:0]	Reserved for discrete feature bits.	N/A

NOTES:

1. Earlier versions of this specification documented these bits as enumerating support for different vector lengths. Processors enumerating Intel® AVX10 support all vector widths.

Several important principles of Intel AVX10 enumeration are the following:

- Versions will be inclusive such that version N+1 is a superset of version N. Once an instruction is introduced in Intel AVX10.x, it is expected to be carried forward in all subsequent Intel AVX10 versions, allowing a developer to check only for a version greater than or equal to the desired version.
- Any processor that enumerates support for Intel AVX10 will also enumerate support for Intel AVX, Intel AVX2, and Intel AVX-512 (see Table 16-2).

The first version of Intel AVX10 (Version 1, or Intel® AVX10.1) supports the Intel AVX-512 instruction families shown in Table 16-2.

Table 16-2. Intel® AVX-512 CPUID Feature Flags Included in Intel® AVX10

Feature Introduction	Intel® AVX-512 CPUID Feature Flags Included in Intel® AVX10
Intel® Xeon® Scalable Processor Family based on Skylake microarchitecture	AVX512F, AVX512CD, AVX512BW, AVX512DQ
Intel® Core™ processors based on Cannon Lake microarchitecture	AVX512-VBMI, AVX512-IFMA
2nd generation Intel® Xeon® Scalable Processor Family based on Cascade Lake product	AVX512-VNNI
3rd generation Intel® Xeon® Scalable Processor Family based on Cooper Lake product	AVX512-BF16
3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture	AVX512-VPOPCNTDQ, AVX512-VBMI2, VAES, GFNI, VPCLMULQDQ, AVX512-BITALG
4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture	AVX512-FP16

NOTE

VAES, VPCLMULQDQ, and GFNI EVEX instructions will be supported on Intel AVX10.1 machines but will continue to be enumerated by their existing discrete CPUID feature flags. This requires the developer to check for both the feature and Intel AVX10, e.g., {AVX10.1 AND VAES}.

New vector ISA features will only be added to the Intel AVX10 ISA moving forward.

4. Updates to Appendix A, Volume 1

Change bars and violet text show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

Changes to this chapter:

- Updated Table A-2, “EFLAGS Cross-Reference” for BSF/BSR.

APPENDIX A EFLAGS CROSS-REFERENCE

A.1 EFLAGS AND INSTRUCTIONS

Table A-2 summarizes how the instructions affect the flags in the EFLAGS register. The following codes describe how the flags are affected.

Table A-1. Codes Describing Flags

T	Instruction tests flag.
M	Instruction modifies flag (either sets or resets depending on operands).
0	Instruction resets flag.
1	Instruction sets flag.
—	Instruction's effect on flag is undefined.
R	Instruction restores prior value of flag.
Blank	Instruction does not affect flag.

Table A-2. EFLAGS Cross-Reference

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
AAA	—	—	—	TM	—	M					
AAD	—	M	M	—	M	—					
AAM	—	M	M	—	M	—					
AAS	—	—	—	TM	—	M					
ADC	M	M	M	M	M	TM					
ADD	M	M	M	M	M	M					
AND	0	M	M	—	M	0					
ARPL			M								
BOUND											
BSF/BSR	0	0	M	0	M	0					
BSWAP											
BT/BTS/BTR/BTC	—	—		—	—	M					
CALL											
CBW											
CLC						0					
CLD									0		
CLI								0			
CLTS											
CMC						M					
CMOV*cc*	T	T	T		T	T					
CMP	M	M	M	M	M	M					

Table A-2. EFLAGS Cross-Reference (Contd.)

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
CMPS	M	M	M	M	M	M			T		
CMPXCHG	M	M	M	M	M	M					
CMPXCHG8B			M								
COMISD	0	0	M	0	M	M					
COMISS	0	0	M	0	M	M					
CPUID											
CWD											
DAA	—	M	M	TM	M	TM					
DAS	—	M	M	TM	M	TM					
DEC	M	M	M	M	M						
DIV	—	—	—	—	—	—					
ENTER											
ESC											
FCMOV*cc*			T		T	T					
FCOMI, FCOMIP, FUCOMI, FUCOMIP	0	0	M	0	M	M					
HLT											
IDIV	—	—	—	—	—	—					
IMUL	M	—	—	—	—	M					
IN											
INC	M	M	M	M	M						
INS									T		
INT							0			0	
INTO	T						0			0	
INVD											
INVLPG											
UCOMISD	0	0	M	0	M	M					
UCOMISS	0	0	M	0	M	M					
IRET	R	R	R	R	R	R	R	R	R	T	
J*cc*	T	T	T		T	T					
JCXZ											
JMP											
LAHF											
LAR			M								
LDS/LES/LSS/LFS/LGS											
LEA											
LEAVE											
LGDT/LIDT/LLDT/LMSW											
LOCK											

Table A-2. EFLAGS Cross-Reference (Contd.)

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
LODS									T		
LOOP											
LOOPE/LOOPNE			T								
LSL			M								
LTR											
MONITOR											
MWAIT											
MOV											
MOV control, debug, test	—	—	—	—	—	—					
MOVS									T		
MOVSX/MOVZX											
MUL	M	—	—	—	—	M					
NEG	M	M	M	M	M	M					
NOP											
NOT											
OR	0	M	M	—	M	0					
OUT											
OUTS									T		
POP/POPA											
POPF	R	R	R	R	R	R	R	R	R	R	
PUSH/PUSHA/PUSHF											
RCL/RCR 1	M					TM					
RCL/RCR count	—					TM					
RDMSR											
RDPMC											
RDTSC											
REP/REPE/REPNE											
RET											
ROL/ROR 1	M					M					
ROL/ROR count	—					M					
RSM	M	M	M	M	M	M	M	M	M	M	M
SAHF		R	R	R	R	R					
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M					
SAL/SAR/SHL/SHR count	—	M	M	—	M	M					
SBB	M	M	M	M	M	TM					
SCAS	M	M	M	M	M	M			T		
SET*cc*	T	T	T		T	T					
SGDT/SIDT/SLDT/SMSW											

Table A-2. EFLAGS Cross-Reference (Contd.)

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
SHLD/SHRD	—	M	M	—	M	M					
STC						1					
STD									1		
STI								1			
STOS									T		
STR											
SUB	M	M	M	M	M	M					
TEST	0	M	M	—	M	0					
UD											
VERR/VERRW			M								
WAIT											
WBINVD											
WRMSR											
XADD	M	M	M	M	M	M					
XCHG											
XLAT											
XOR	0	M	M	—	M	0					

5. Updates to Chapter 2, Volume 2A

Change bars and violet text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A:* Instruction Set Reference, A-L.

Changes to this chapter:

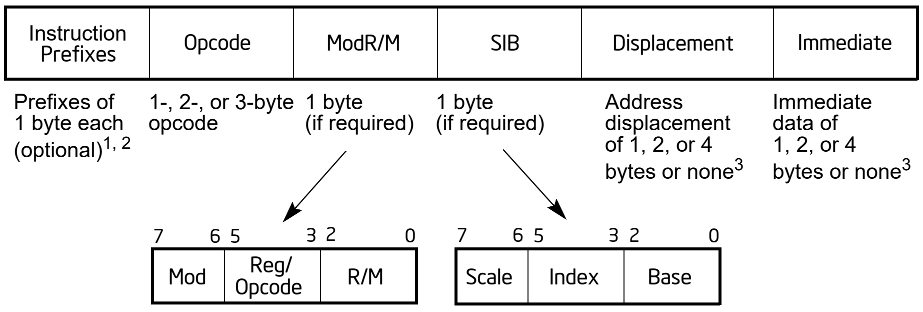
- Corrected exception type for instructions VCVTPS2PD, VCVTPS2QQ, VCVTPS2UQQ, VCVTTPS2QQ, and VCVTTPS2UQQ from Type E3 to Type E2 in Table 2-45, "EVEX Instructions in Each Exception Class" in Section 2.8, "Exception Classifications of EVEX-Encoded instructions."

CHAPTER 2
INSTRUCTION FORMAT

This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, “REX Prefixes” for additional information.
2. For VEX encoding information, see Section 2.3, “Intel® Advanced Vector Extensions (Intel® AVX)”.
3. Some rare instructions can take an 8B immediate or 8B displacement.

Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- Group 1
  - — Lock and repeat prefixes:
    - LOCK prefix is encoded using F0H.
    - REPNE/REPNZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. (F3H is also used as a mandatory prefix for some instructions.)

- — BND prefix is encoded using F2H if the following conditions are true:
  - CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.
  - BNDCFGU.EN and/or IA32_BNDCFGS.EN is set.
  - When the F2 prefix precedes a near CALL, a near RET, a near JMP, a short Jcc, or a near Jcc instruction (see Appendix E, "Intel® Memory Protection Extensions," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- Group 2
  - — Segment override prefixes:
    - 2EH—CS segment override (use with any branch instruction is reserved).
    - 36H—SS segment override prefix (use with any branch instruction is reserved).
    - 3EH—DS segment override prefix (use with any branch instruction is reserved).
    - 26H—ES segment override prefix (use with any branch instruction is reserved).
    - 64H—FS segment override prefix (use with any branch instruction is reserved).
    - 65H—GS segment override prefix (use with any branch instruction is reserved).
  - — Branch hints[1]:
    - 2EH—Branch not taken (used only with J*cc* instructions).
    - 3EH—Branch taken (used only with J*cc* instructions).
- Group 3
  - Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).
- Group 4
  - 67H—Address-size override prefix.

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See "LOCK—Assert LOCK# Signal Prefix" in Chapter 3, "Instruction Set Reference, A-L," for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string and I/O instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Some instructions may use F2H or F3H as a mandatory prefix to express distinct functionality.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch when used on conditional branch instructions (J*cc*).

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality.

Other use of the 66H prefix is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

1. Microarchitectural behavior varies; refer to the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet).

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet).

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The *mod* field combines with the r/m field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or it can be combined with the mod field to encode an addressing mode. Sometimes, certain combinations of the *mod* field and the *r/m* field are used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled “REG =”). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute.

If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled “/digit (Opcode)”). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label “Value of ModR/M Byte (in Hexadecimal)”) contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4, and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1, and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

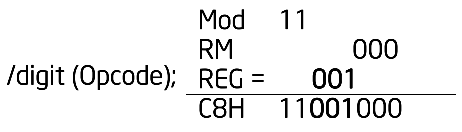


Figure 2-2. Table Interpretation of ModR/M Byte (C8H)

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
(In decimal) /digit (Opcode)
(In binary) REG =
			AL
			AX
			EAX
			MM0
			XMM0
			0
			000
				CL
				CX
				ECX
				MM1
				XMM1
				1
				001
					DL
					DX
					EDX
					MM2
					XMM2
					2
					010
						BL
						BX
						EBX
						MM3
						XMM3
						3
						011
							AH
							SP
							ESP
							MM4
							XMM4
							4
							100
								CH
								BP[1]
								EBP
								MM5
								XMM5
								5
								101
									DH
									SI
									ESI
									MM6
									XMM6
									6
									110
										BH
										DI
										EDI
										MM7
										XMM7
										7
										111

Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AHMM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
(In decimal) /digit (Opcode)
(In binary) REG =
			AL
			AX
			EAX
			MM0
			XMM0
			0
			000
				CL
				CX
				ECX
				MM1
				XMM1
				1
				001
					DL
					DX
					EDX
					MM2
					XMM2
					2
					010
						BL
						BX
						EBX
						MM3
						XMM3
						3
						011
							AH
							SP
							ESP
							MM4
							XMM4
							4
							100
								CH
								BP
								EBP
								MM5
								XMM5
								5
								101
									DH
									SI
									ESI
									MM6
									XMM6
									6
									110
										BH
										DI
										EDI
										MM7
										XMM7
										7
										111

Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4, and 5) and the scaling factor (determined by SIB byte bits 6 and 7).

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:

- **Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- **64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.

2.2.1 REX Prefixes

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.

- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A REX prefix is necessary only if an instruction references one of the extended registers or one of the byte registers SPL, BPL, SIL, DIL; or uses a 64-bit operand. A REX prefix is ignored, as are its individual bits, when it is not needed for an instruction or when it does not immediately precede the opcode byte or the escape opcode byte (0FH) of an instruction for which it is needed. This has the implication that only one REX prefix, properly located, can affect an instruction.

When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can immediately precede the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

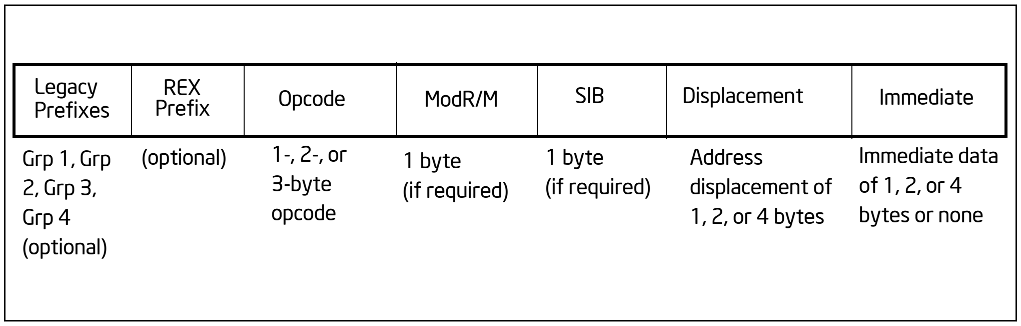


Figure 2-3. Prefix Ordering in 64-bit Mode

2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 though Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.

- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

Table 2-4. REX Prefix Fields [BITS: 0100WRXB]

Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D
		1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field

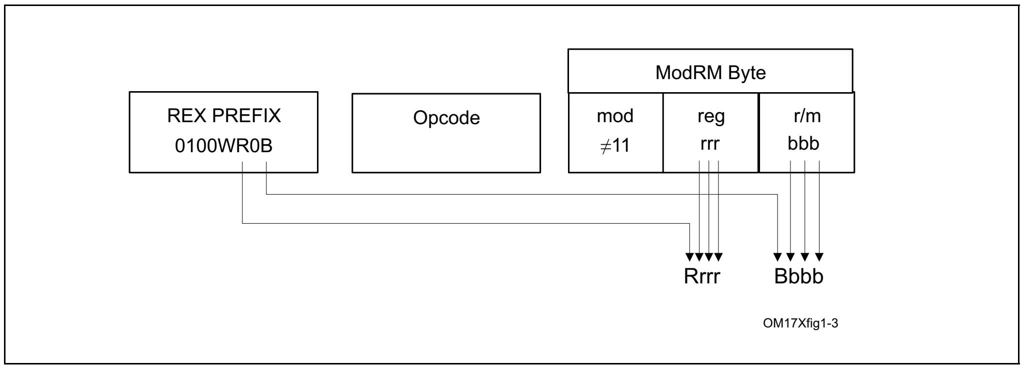


Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

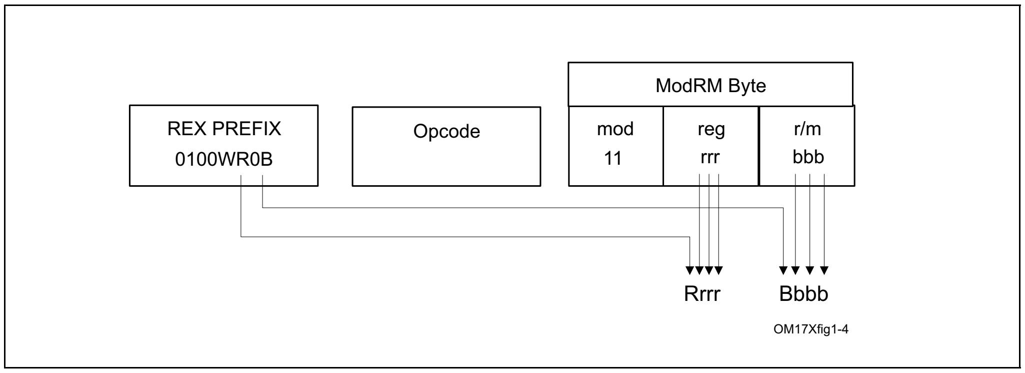


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

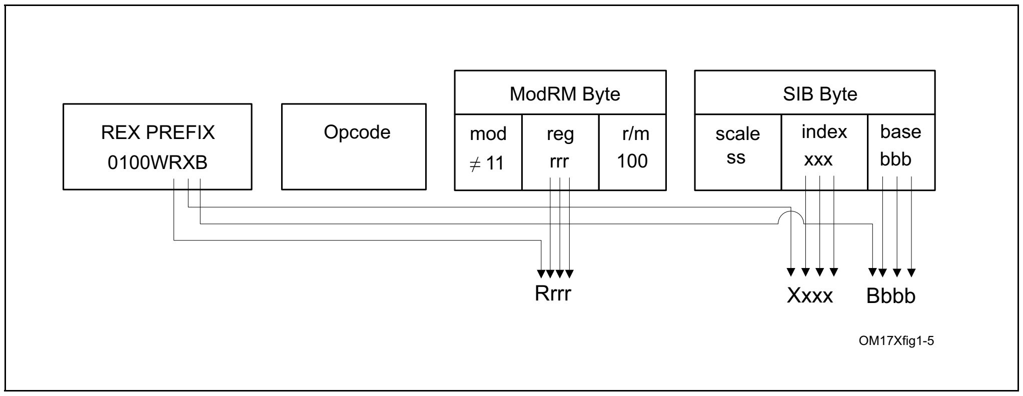


Figure 2-6. Memory Addressing With a SIB Byte

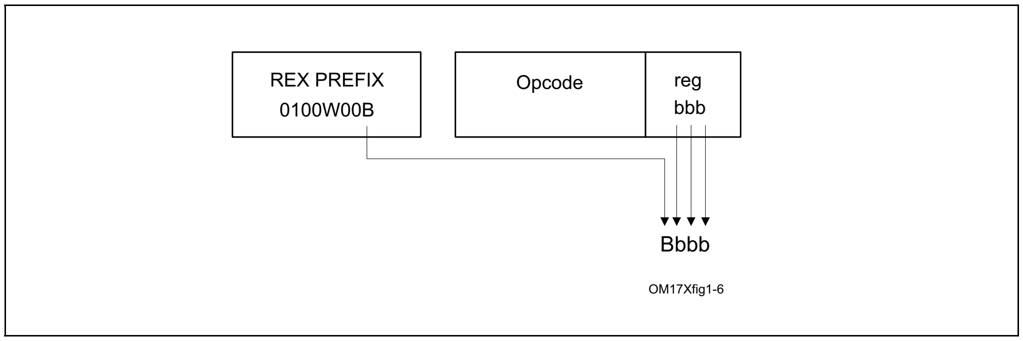


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte's reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations.

Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.

Table 2-5. Special Cases of REX Encodings

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
ModR/M Byte	mod ? 11 r/m = b*100(ESP)	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care). SIB byte also required for R12-based addressing.
ModR/M Byte	mod = 0 r/m = b*101(EBP)	Base register not used.	EBP without a displacement must be done using mod = 01 with displacement of 0.	REX prefix adds a fourth bit (b) which is not decoded (don't care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.
SIB Byte	index = 0100(ESP)	Index register not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.

Table 2-5. Special Cases of REX Encodings (Contd.)

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
SIB Byte	base = 0101(EBP)	Base register is unused if mod = 0.	Base register depends on mod encoding.	REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.

NOTES:

* Don't care about value of REX.B

2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

Table 2-6. Direct Memory Offset Form of MOV

Opcode	Instruction
A0	MOV AL, moffset
A1	MOV EAX, moffset
A2	MOV moffset, AL
A3	MOV moffset, EAX

2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

```
48 B8  8877665544332211  MOV RAX,1122334455667788H
```

2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

Table 2-7. RIP-Relative Addressing

ModR/M and SIB Sub-field Encodings		Compatibility Mode Operation	64-bit Mode Operation	Additional Implications in 64-bit mode
ModR/M Byte	mod = 00	Disp32	RIP + Disp32	In 64-bit mode, if one wants to use a Disp32 without specifying a base register, one can use a SIB byte encoding (indicated by ModR/M.r/m=100) as described in the next row.
	r/m = 101 (none)			
SIB Byte	base = 101 (none)	If mod = 00, Disp32	Same as legacy	None
	index = 100 (none)			
	scale = 0, 1, 2, 4			

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8- DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).

2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel AVX instructions are encoded using an encoding scheme that combines prefix bytes, opcode extension field, operand encoding fields, and vector length encoding capability into a new prefix, referred to as VEX. In the VEX encoding scheme, the VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the two-byte or three-byte length of the VEX prefix, the VEX encoding format provides a more compact representation/packing of the components of encoding an instruction in Intel 64 architecture. The VEX encoding scheme also allows more headroom for future growth of Intel 64 architecture.

2.3.1 Instruction Format

Instruction encoding using VEX prefix provides several advantages:

- Instruction syntax support for three operands and up-to four operands when necessary. For example, the third source register used by VBLENDVPD is encoded using bits 7:4 of the immediate byte.
- Encoding support for vector length of 128 bits (using XMM registers) and 256 bits (using YMM registers).
- Encoding support for instruction syntax of non-destructive source operands.
- Elimination of escape opcode byte (0FH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
- Extensibility for future instruction extensions without significant instruction length increase.

Figure 2-8 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, “single-purpose” byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

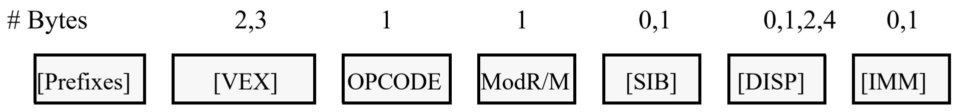


Figure 2-8. Instruction Encoding Format with VEX Prefix

2.3.2 VEX and the LOCK prefix

Any VEX-encoded instruction with a LOCK prefix preceding VEX will #UD.

2.3.3 VEX and the 66H, F2H, and F3H prefixes

Any VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

2.3.4 VEX and the REX prefix

Any VEX-encoded instruction with a REX prefix proceeding VEX will #UD.

2.3.5 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- Non-destructive source register encoding (applicable to three and four operand syntax): This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1's complement form (inverted form), i.e., XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
- Vector length encoding: This 1-bit field represented by the notation VEX.L. L= 0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.
- REX prefix functionality: Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1's complement form, i.e., XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
    - Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1's complement encoding. This is represented as VEX.R.
    - Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1's complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
    - Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

  Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.
- Compaction of SIMD prefix: Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.
- Compaction of two-byte and three-byte opcode: More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

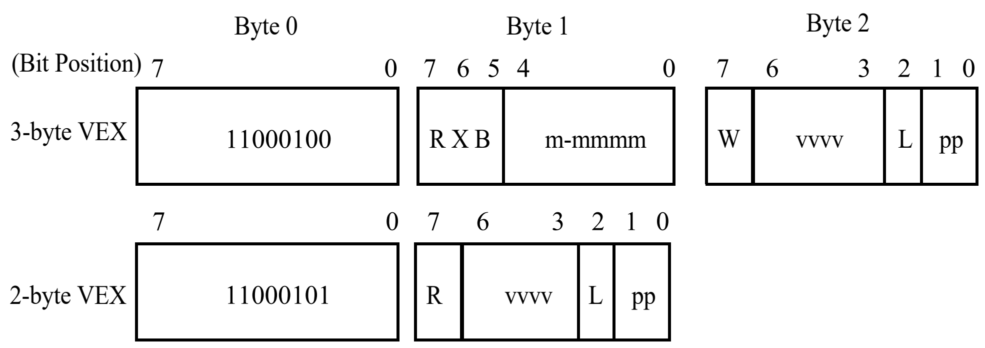
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.



R: REX.R in 1's complement (inverted) form
  1: Same as REX.R=0 (must be 1 in 32-bit mode)
  0: Same as REX.R=1 (64-bit mode only)
X: REX.X in 1's complement (inverted) form
  1: Same as REX.X=0 (must be 1 in 32-bit mode)
  0: Same as REX.X=1 (64-bit mode only)
B: REX.B in 1's complement (inverted) form
  1: Same as REX.B=0 (Ignored in 32-bit mode).
  0: Same as REX.B=1 (64-bit mode only)
W: opcode specific (use like REX.W, or used for opcode extension, or ignored, depending on the opcode byte)

m-mmmm:
  00000: Reserved for future use (will #UD)
  00001: implied 0F leading opcode byte
  00010: implied 0F 38 leading opcode bytes
  00011: implied 0F 3A leading opcode bytes
  00100-11111: Reserved for future use (will #UD)

vvvv: a register specifier (in 1's complement form) or 1111 if unused.

L: Vector Length
  0: scalar or 128-bit vector
  1: 256-bit vector

pp: opcode extension providing equivalent functionality of a SIMD prefix
  00: None
  01: 66
  10: F3
  11: F2

Figure 2-9. VEX bit fields

The following subsections describe the various fields in two or three-byte VEX prefix.

2.3.5.1 VEX Byte 0, bits[7:0]

VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

2.3.5.2 VEX Byte 1, bit [7] - 'R'

VEX Byte 1, bit [7] contains a bit analogous to a bit inverted REX.R. In protected and compatibility modes the bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is present in both 2- and 3-byte VEX prefixes.

The usage of WRXB bits for legacy instructions is explained in detail section 2.2.1.2 of Intel 64 and IA-32 Architectures Software developer's manual, Volume 2A.

This bit is stored in bit inverted format.

2.3.5.3 3-byte VEX byte 1, bit[6] - 'X'

Bit[6] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.X. It is an extension of the SIB Index field in 64-bit modes. In 32-bit modes, this bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

2.3.5.4 3-byte VEX byte 1, bit[5] - 'B'

Bit[5] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.B. In 64-bit modes, it is an extension of the ModR/M r/m field, or the SIB base field. In 32-bit modes, this bit is ignored.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

2.3.5.5 3-byte VEX byte 2, bit[7] - 'W'

Bit[7] of the 3-byte VEX byte 2 is represented by the notation VEX.W. It can provide following functions, depending on the specific opcode.

- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes for these instructions, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don't care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or Dest Register Specifier

In 32-bit mode the VEX first byte C4 and C5 alias onto the LES and LDS instructions. To maintain compatibility with existing programs the VEX 2nd byte, bits [7:6] must be 11b. To achieve this, the VEX payload bits are selected to place only inverted, 64-bit valid fields (extended register selectors) in these upper bits.

The 2-byte VEX Byte 1, bits [6:3] and the 3-byte VEX, Byte 2, bits [6:3] encode a field (shorthand VEX.vvvv) that for instructions with 2 or more source registers and an XMM or YMM or memory destination encodes the first source register specifier stored in inverted (1's complement) form.

VEX.vvvv is not used by the instructions with one source (except certain shifts, see below) or on instructions with no XMM or YMM or memory destination. If an instruction does not use VEX.vvvv then it should be set to 1111b otherwise instruction will #UD.

In 64-bit mode all 4 bits may be used. See Table for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).

Table 2-8. VEX.vvvv to Register Name Mapping

VEX.vvvv	Dest Register	General-Purpose Register (If Applicable)[1]	Valid in Legacy/Compatibility 32-bit modes?[2]
1111B	XMM0/YMM0	RAX/EAX	Valid
1110B	XMM1/YMM1	RCX/ECX	Valid
1101B	XMM2/YMM2	RDX/EDX	Valid
1100B	XMM3/YMM3	RBX/EBX	Valid
1011B	XMM4/YMM4	RSP/ESP	Valid
1010B	XMM5/YMM5	RBP/EBP	Valid
1001B	XMM6/YMM6	RSI/ESI	Valid
1000B	XMM7/YMM7	RDI/EDI	Valid
0111B	XMM8/YMM8	R8/R8D	Invalid
0110B	XMM9/YMM9	R9/R9D	Invalid
0101B	XMM10/YMM10	R10/R10D	Invalid
0100B	XMM11/YMM11	R11/R11D	Invalid
0011B	XMM12/YMM12	R12/R12D	Invalid
0010B	XMM13/YMM13	R13/R13D	Invalid
0001B	XMM14/YMM14	R14/R14D	Invalid
0000B	XMM15/YMM15	R15/R15D	Invalid

NOTES:

1. See Section 2.6, "VEX Encoding Support for GPR Instructions" for additional details.
2. Only the first eight General-Purpose Registers are accessible/encodable in 16/32b modes.

The VEX.vvvv field is encoded in bit inverted format for accessing a register operand.

2.3.6 Instruction Operand Encoding and VEX.vvvv, ModR/M

VEX-encoded instructions support three-operand and four-operand instruction syntax. Some VEX-encoded instructions have syntax with less than three operands, e.g., VEX-encoded pack shift instructions support one source operand and one destination operand).

The roles of VEX.vvvv, reg field of ModR/M byte (ModR/M.reg), r/m field of ModR/M byte (ModR/M.r/m) with respect to encoding destination and source operands vary with different type of instruction syntax.

The role of VEX.vvvv can be summarized to three situations:

- VEX.vvvv encodes the first source register operand, specified in inverted (1's complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1's complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 2-9. The notation in the "Opcode" column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 2-9. Instructions with a VEX.vvvv Destination

Opcode	Instruction mnemonic
VEX.128.66.0F 73 /7 ib	VPSLLDQ xmm1, xmm2, imm8
VEX.128.66.0F 73 /3 ib	VPSRLDQ xmm1, xmm2, imm8

Table 2-9. Instructions with a VEX.vvvv Destination (Contd.)

Opcode	Instruction mnemonic
VEX.128.66.0F 71 /2 ib	VPSRLW xmm1, xmm2, imm8
VEX.128.66.0F 72 /2 ib	VPSRLD xmm1, xmm2, imm8
VEX.128.66.0F 73 /2 ib	VPSRLQ xmm1, xmm2, imm8
VEX.128.66.0F 71 /4 ib	VPSRAW xmm1, xmm2, imm8
VEX.128.66.0F 72 /4 ib	VPSRAD xmm1, xmm2, imm8
VEX.128.66.0F 71 /6 ib	VPSLLW xmm1, xmm2, imm8
VEX.128.66.0F 72 /6 ib	VPSLLD xmm1, xmm2, imm8
VEX.128.66.0F 73 /6 ib	VPSLLQ xmm1, xmm2, imm8

The role of ModR/M.r/m field can be summarized to two situations:

- ModR/M.r/m encodes the instruction operand that references a memory address.
- For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:

- ModR/M.reg encodes either the destination register operand or a source register operand.
- For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves the following situation:

- Imm8[7:4] encodes the third source register operand.

2.3.6.1 3-byte VEX byte 1, bits[4:0] - “m-mmmm”

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

Table 2-10. VEX.m-mmmm Interpretation

VEX.m-mmmm	Implied Leading Opcode Bytes
00000B	Reserved
00001B	0F
00010B	0F 38
00011B	0F 3A
00100-11111B	Reserved
(2-byte VEX)	0F

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- “L”

The vector length field, VEX.L, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If “VEX.L = 1”, it indicates 256-bit vector operation. “VEX.L = 0” indicates scalar and 128-bit vector operations.

The instruction VZEROUPPER is a special case that is encoded with VEX.L = 0, although its operation zero’s bits 255:128 of all YMM registers accessible in the current operating mode. See Table 2-11.

Table 2-11. VEX.L Interpretation

VEX.L	Vector Length
0	128-bit (or 32/64-bit scalar)
1	256-bit

2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- “pp”

Up to one implied prefix is encoded by bits[1:0] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes. See Table 2-12.

Table 2-12. VEX.pp Interpretation

pp	Implies this prefix after other prefixes but before VEX
00B	None
01B	66
10B	F3
11B	F2

2.3.7 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

2.3.8 The ModR/M, SIB, and Displacement Bytes

The encodings are unchanged but the interpretation of reg_field or rm_field differs (see above).

2.3.9 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

2.3.10 Intel® AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper bits (above bit 128) of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper bits.

2.3.10.1 Vector Length Transition and Programming Considerations

An instruction encoded with a VEX.128 prefix that loads a YMM register operand operates as follows:

- Data is loaded into bits 127:0 of the register
- Bits above bit 127 in the register are cleared.

Thus, such an instruction clears bits 255:128 of a destination YMM register on processors with a maximum vector-register width of 256 bits. In the event that future processors extend the vector registers to greater widths, an instruction encoded with a VEX.128 or VEX.256 prefix will also clear any bits beyond bit 255. (This is in contrast with legacy SSE instructions, which have no VEX prefix; these modify only bits 127:0 of any destination register operand.)

Programmers should bear in mind that instructions encoded with VEX.128 and VEX.256 prefixes will clear any future extensions to the vector registers. A calling function that uses such extensions should save their state before calling legacy functions. This is not possible for involuntary calls (e.g., into an interrupt-service routine). It is recommended that software handling involuntary calls accommodate this by not executing instructions encoded with VEX.128 and VEX.256 prefixes. In the event that it is not possible or desirable to restrict these instructions, then software must take special care to avoid actions that would, on future processors, zero the upper bits of vector registers.

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

2.3.11 Intel® AVX Instruction Length

The Intel AVX instructions described in this document (including VEX and ignoring other prefixes) do not exceed 11 bytes in length, but may increase in the future. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes.

2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® AVX2), an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of Intel AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-13 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. The register names also include R8D-R15D applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-13). In 32-bit mode, R8D-R15D does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The left-most column includes vector registers VR8-VR15 (i.e., XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte

r32 (In decimal) Base = (In binary) Base =				EAX/ R8D 0 000	ECX/ R9D 1 001	EDX/ R10D 2 010	EBX/ R11D 3 011	ESP/ R12D 4 100	EBP/ R13D[1] 5 101	ESI/ R14D 6 110	EDI/ R15D 7 111
Scaled Index		**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
VR0/VR8	*1	00	000	00	01	02	03	04	05	06	07
VR1/VR9			001	08	09	0A	0B	0C	0D	0E	0F
VR2/VR10			010	10	11	12	13	14	15	16	17
VR3/VR11			011	18	19	1A	1B	1C	1D	1E	1F
VR4/VR12			100	20	21	22	23	24	25	26	27
VR5/VR13			101	28	29	2A	2B	2C	2D	2E	2F
VR6/VR14			110	30	31	32	33	34	35	36	37
VR7/VR15			111	38	39	3A	3B	3C	3D	3E	3F
VR0/VR8	*2	01	000	40	41	42	43	44	45	46	47
VR1/VR9			001	48	49	4A	4B	4C	4D	4E	4F
VR2/VR10			010	50	51	52	53	54	55	56	57
VR3/VR11			011	58	59	5A	5B	5C	5D	5E	5F
VR4/VR12			100	60	61	62	63	64	65	66	67
VR5/VR13			101	68	69	6A	6B	6C	6D	6E	6F
VR6/VR14			110	70	71	72	73	74	75	76	77
VR7/VR15			111	78	79	7A	7B	7C	7D	7E	7F
VR0/VR8	*4	10	000	80	81	82	83	84	85	86	87
VR1/VR9			001	88	89	8A	8B	8C	8D	8E	8F
VR2/VR10			010	90	91	92	93	94	95	96	97
VR3/VR11			011	98	89	9A	9B	9C	9D	9E	9F
VR4/VR12			100	A0	A1	A2	A3	A4	A5	A6	A7
VR5/VR13			101	A8	A9	AA	AB	AC	AD	AE	AF
VR6/VR14			110	B0	B1	B2	B3	B4	B5	B6	B7
VR7/VR15			111	B8	B9	BA	BB	BC	BD	BE	BF
VR0/VR8	*8	11	000	C0	C1	C2	C3	C4	C5	C6	C7
VR1/VR9			001	C8	C9	CA	CB	CC	CD	CE	CF
VR2/VR10			010	D0	D1	D2	D3	D4	D5	D6	D7
VR3/VR11			011	D8	D9	DA	DB	DC	DD	DE	DF
VR4/VR12			100	E0	E1	E2	E3	E4	E5	E6	E7
VR5/VR13			101	E8	E9	EA	EB	EC	ED	EE	EF
VR6/VR14			110	F0	F1	F2	F3	F4	F5	F6	F7
VR7/VR15			111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. If ModR/M.mod = 00b, the base address is zero, then effective address is computed as [scaled vector index] + disp32. Otherwise the base address is computed as [EBP/R13]+ disp, the displacement is either 8 bit or 32 bit depending on the value of ModR/M.mod:

MOD	Effective Address
00b	[Scaled Vector Register] + Disp32
01b	[Scaled Vector Register] + Disp8 + [EBP/R13]
10b	[Scaled Vector Register] + Disp32 + [EBP/R13]

2.3.12.1 64-bit Mode VSIB Memory Addressing

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose register as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 2-13 base register should be interpreted as the full 64-bit of each register.

2.4 INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

Intel® AMX instructions follow the general documentation convention established in previous sections. Additionally, Intel® Advanced Matrix Extensions use notation conventions as described below.

In the instruction encoding boxes, **sibmem** is used to denote an encoding where a ModR/M byte and SIB byte are used to indicate a memory operation where the base and displacement are used to point to memory, and the index register (if present) is used to denote a stride between memory rows. The index register is scaled by the sib.scale field as usual. The base register is added to the displacement, if present.

In the instruction encoding, the ModR/M byte is represented several ways depending on the role it plays. The ModR/M byte has 3 fields: 2-bit ModR/M.mod field, a 3-bit ModR/M.reg field and a 3-bit ModR/M.r/m field. When all bits of the ModR/M byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the ModR/M byte must contain fixed values, those values are specified as follows:

- If only the ModR/M.mod must be 0b11, and ModR/M.reg and ModR/M.r/m fields are unrestricted, this is denoted as **11:rrr:bbb**. The **rrr** correspond to the 3-bits of the ModR/M.reg field and the **bbb** correspond to the 3-bits of the ModR/M.r/m field.
- If the ModR/M.mod field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then the notation !(11) is used.
- If the ModR/M.reg field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

NOTE

Historically this document only specified the ModR/M.reg field restrictions with the notation /0 ... /7 and did not specify restrictions on the ModR/M.mod and ModR/M.r/m fields in the encoding boxes.

2.5 INTEL® AVX AND INTEL® SSE INSTRUCTION EXCEPTION CLASSIFICATION

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.5.1 through 2.6.1. For example, ADDPS contains the entry:

"See Exceptions Type 2."

In this entry, "Type 2" can be looked up in Table 2-19.

The instruction's corresponding CPUID feature flag can be identified in the fourth column of the Instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

NOTE

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Table 2-14. Exception Class Description

Exception Class	Instruction Set	Mem Arg	Floating-Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	No
Type 2	AVX, Legacy SSE	16/32 byte not explicitly aligned	Yes
Type 3	AVX, Legacy SSE	< 16 byte	Yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned	No
Type 5	AVX, Legacy SSE	< 16 byte	No
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)

Table 2-14. Exception Class Description (Contd.)

Exception Class	Instruction Set	Mem Arg	Floating-Point Exceptions (#XM)
Type 7	AVX, Legacy SSE	None	No
Type 8	AVX	None	No
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	Yes
Type 12	AVX2 Gathers	Not explicitly aligned, no AC#	No

See Table 2-15 for lists of instructions in each exception class.

Table 2-15. Instructions in Each Exception Class

Exception Class	Instruction
Type 1	(V)MOVAPD, (V)MOVAPS, (V)MOVDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS
Type 2	(V)ADDPD, (V)ADDPS, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ, (V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS, VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPD, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS
Type 3	(V)ADDSD, (V)ADDSS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTTSD2SI, (V)CVTTSS2SI, (V)DIVSD, (V)DIVSS, VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADD132SS, VFMADD213SS, VFMADD231SS, VFMSUB132SD, VFMSUB213SD, VFMSUB231SD, VFMSUB132SS, VFMSUB213SS, VFMSUB231SS, VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS, (V)MAXSD, (V)MAXSS, (V)MINSD, (V)MINSS, (V)MULSD, (V)MULSS, (V)ROUNDSD, (V)ROUNDSS, (V)SQRTSD, (V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMISD, (V)UCOMISS
Type 4	(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, VBLENDVPD, VBLENDVPS, (V)LDDQU***, (V)MASKMOVDQU, (V)PTEST, VTESTPS, VTESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB, (V)PBLENDW, (V)PCMP(E/I)STRI/M***, (V)PCMPEQB, (V)PCMPEQW, (V)PCMPEQD, (V)PCMPEQQ, (V)PCMPGTB, (V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADDD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, (V)PMADDWD, (V)PMADDUBSW, (V)PMAXSB, (V)PMAXSW, (V)PMAXSD, (V)PMAXUB, (V)PMAXUW, (V)PMAXUD, (V)PMINSB, (V)PMINSW, (V)PMINSD, (V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRSW, (V)PMULHW, (V)PMULLW, (V)PMULLD, (V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUFB, (V)PSHUFD, (V)PSHUFHW, (V)PSHUFLW, (V)PSIGNB, (V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ, (V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSB, (V)PSUBUSW, (V)PUNPCKHBW, (V)PUNPCKHWD, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS, (V)XORPD, (V)XORPS, VPBLENDD, VPERMD, VPERMPS, VPERMPD, VPERMQ, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ, VPERMILPD, VPERMILPS, VPERM2F128

Table 2-15. Instructions in Each Exception Class (Contd.)

Exception Class	Instruction
Type 5	(V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPS, (V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ, PMOVSXBW, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX, VLDMXCSR*, VSTMXCSR
Type 6	VEXTRACTF128/VEXTRACTFxxxx, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS**, VMASKMOVPD**, VPMASKMOVD, VPMASKMOVQ, VBROADCASTI128, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VEXTRACTI128, VINSERTI128, VPERM2I128
Type 7	(V)MOVLHPS, (V)MOVHLPS, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMSKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ
Type 8	VZEROALL, VZEROUPPER
Type 11	VCVTPH2PS, VCVTPS2PH
Type 12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

(*) - Additional exception restrictions are present - see the Instruction description for details

(**) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e., no alignment checks are performed.

(***) - PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM, and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for Intel AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of Intel AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-16 and Table 2-17 provide supplemental information of Intel AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

Table 2-16. #UD Exception and VEX.W=1 Encoding

Exception Class	#UD If VEX.W = 1 in All Modes	#UD If VEX.W = 1 in Non-64-Bit Modes
Type 1		
Type 2		
Type 3		
Type 4	VBLENDVPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLENDD, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMILPD, VPERMILPS, VPERM2F128	
Type 5		
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSERTI128	
Type 7		
Type 8		
Type 11	VCVTPH2PS, VCVTPS2PH	
Type 12		

Table 2-17. #UD Exception and VEX.L Field Encoding

Exception Class	#UD If VEX.L = 0	#UD If (VEX.L = 1 && AVX2 not present && AVX present)	#UD If (VEX.L = 1 && AVX2 present)
Type 1		VMOVNTDQA	
Type 2		VDPPD	VDPPD
Type 3			
Type 4		VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMAXSB/W/D, VPMAXUB/W/D, VPMINSB/W/D, VPMINUB/W/D, VPMULHUW, VPMULHRSW, VPMULHW/LW, VPMULLD, VPMULUDQ, VPMULDQ, VPOR, VPSADBW, VPSHUFB/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/WD/DQ, VPUNPCKHQDQ, VPUNPCKLBW/WD/DQ, VPUNPCKLQDQ, VPXOR	VPCMP(E/I)STRI/M, PHMINPOSUW
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX, VLDMXCSR, VSTMXCSR	Same as column 3
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,		
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMSKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ	VMOVLHPS, VMOVHLPS
Type 8			
Type 11			
Type 12			

2.5.1 Exceptions Type 1 (Aligned Memory Reference)

Table 2-18. Type 1 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ? '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	VEX.256: Memory operand is not 32-byte aligned. VEX.128: Memory operand is not 16-byte aligned.
	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.5.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned)

Table 2-19. Type 2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCR0[2:1] ? ‘11b’. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

2.5.3 Exceptions Type 3 (<16 Byte Memory Argument)

Table 2-20. Type 3 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCR0[2:1] ? '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

2.5.4 Exceptions Type 4 (>=16 Byte Mem Arg, No Alignment, No Floating-point Exceptions)

Table 2-21. Type 4 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ? ‘11b’. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.[1]
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

NOTES:

1. LDDQU, MOVUPD, MOVUPS, PCMPESTRI, PCMPESTRM, PCMPISTRI, and PCMPISTRM instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

2.5.5 Exceptions Type 5 (<16 Byte Mem Arg and No FP Exceptions)

Table 2-22. Type 5 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ? '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.5.6 Exceptions Type 6 (VEX-Encoded Instructions without Legacy SSE Analogues)

Note: At present, the AVX instructions in this category do not generate floating-point exceptions.

Table 2-23. Type 6 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	If XCR0[2:1] ? '11b'. If CR4.OSXSAVE[bit 18]=0.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.5.7 Exceptions Type 7 (No FP Exceptions, No Memory Arg)

Table 2-24. Type 7 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ? '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.5.8 Exceptions Type 8 (AVX and No Memory Argument)

Table 2-25. Type 8 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual-8086 mode.
			X	X	If XCR0[2:1] ? '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv ? 1111B.
	X	X	X	X	If proceeded by a LOCK prefix (F0H).
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.5.9 Exceptions Type 11 (VEX-only, Mem Arg, No AC, Floating-point Exceptions)

Table 2-26. Type 11 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ? ‘11b’. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

2.5.10 Exceptions Type 12 (VEX-only, VSIB Mem Arg, No AC, No Floating-point Exceptions)

Table 2-27. Type 12 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ? ‘11b’. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = ‘11b’.
	X	X	X	X	If ModR/M.rm ? ‘100b’.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
	X	X	X	X	If any vector register is used more than once between the destination register, mask register and the index register in VSIB addressing.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

2.6 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

The VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (0FH), two-byte escape via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access or memory addressing.
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only.
- VEX-encoded GPR instructions are encoded with VEX.L=0.

Any VEX-encoded GPR instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

Any VEX-encoded GPR instruction with a REX prefix proceeding VEX will #UD.

VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

2.6.1 Exceptions Type 13 (VEX-Encoded GPR Instructions)

The exception conditions applicable to VEX-encoded GPR instructions differ from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GPR instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

Table 2-28. VEX-Encoded GPR Instructions

Exception Class	Instruction
Type 13	ANDN, BEXTR, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

(*) - Additional exception restrictions are present - see the Instruction description for details.

Table 2-29. Type 13 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If BMI1/BMI2 CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
	X	X	X	X	If VEX.L = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, #SS(0)	X	X	X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.6.2 Exceptions Type 14 (CMPCCXADD)

The exception conditions applicable to the CMPCCXADD instruction differ from those of other VEX-encoded GPR instructions. The exception conditions for the CMPCCXADD instruction are found in Table 2-31.

Table 2-30. Exceptions Type 14 Instructions

Exception Class	Instruction
Type 14	CMPCCXADD

Table 2-31. Type 14 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X		Only supported in 64-bit mode.
				X	If any LOCK, REX, F2, F3, or 66 prefixes precede a VEX prefix.
				X	If any corresponding CPUID feature flag is '0'.
Stack, #SS(0)				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)				X	If the memory address is in a non-canonical form.
Page Fault, #PF(fault-code)				X	If a page fault occurs.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.7 INTEL® AVX-512 ENCODING

The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.

The significant feature differences between EVEX and VEX are summarized below.

- EVEX is a 4-Byte prefix (the first byte must be 62H); VEX is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- EVEX prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- EVEX prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- EVEX memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve the encoding density of the instruction byte stream.
- EVEX prefix can encode functionality that are specific to instruction classes (e.g., packed instruction with "load+op" semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support "suppress all exceptions" functionality).

2.7.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10. Note that the values contained within brackets are optional.

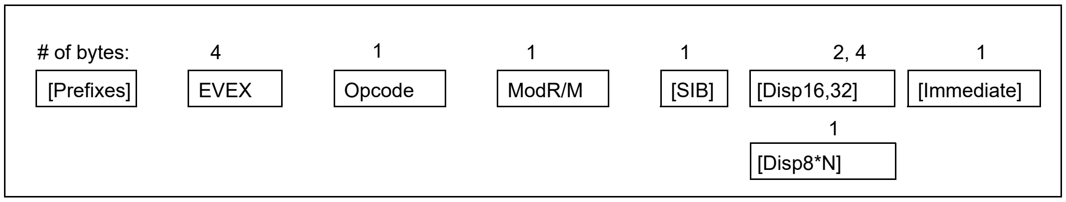
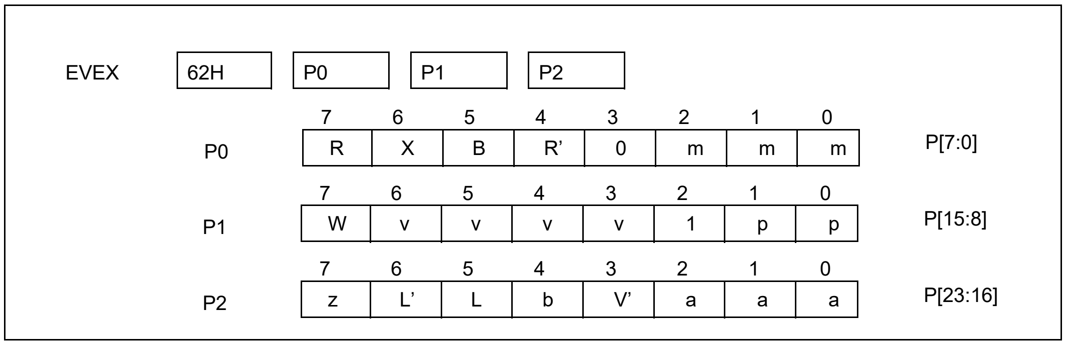


Figure 2-10. Intel® AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).



EVEX
62H
P0
P1
P2
P0: 7 R, 6 X, 5 B, 4 R', 3 0, 2 m, 1 m, 0 m — P[7:0]
P1: 7 W, 6 v, 5 v, 4 v, 3 v, 2 1, 1 p, 0 p — P[15:8]
P2: 7 z, 6 L', 5 L, 4 b, 3 V', 2 a, 1 a, 0 a — P[23:16]

Figure 2-11. Bit Field Layout of the EVEX Prefix[1]

NOTES:

1. See Table 2-32 for additional details on bit fields.

Table 2-32. EVEX Prefix Bit Field Functional Grouping

Notation	Bit field Group	Position	Comment
EVEX.mmm	Access to up to eight decoding maps	P[2:0]	Currently, only the following decoding maps are supported: 1, 2, 3, 5, and 6.
--	Reserved	P[3]	Must be 0.
EVEX.R'	High-16 register specifier modifier	P[4]	Combine with EVEX.R and ModR/M.reg. This bit is stored in inverted format.
EVEX.RXB	Next-8 register specifier modifier	P[7:5]	Combine with ModR/M.reg, ModR/M.rm (base, index/vidx). This field is encoded in bit inverted format.
EVEX.X	High-16 register specifier modifier	P[6]	Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent.
EVEX.pp	Compressed legacy prefix	P[9:8]	Identical to VEX.pp.
--	Fixed Value	P[10]	Must be 1.
EVEX.vvvv	VVVV register specifier	P[14:11]	Same as VEX.vvvv. This field is encoded in bit inverted format.
EVEX.W	Operand size promotion/Opcode extension	P[15]	
EVEX.aaa	Embedded opmask register specifier	P[18:16]	
EVEX.V'	High-16 VVVV/VIDX register specifier	P[19]	Combine with EVEX.vvvv or when VSIB present. This bit is stored in inverted format.
EVEX.b	Broadcast/RC/SAE Context	P[20]	
EVEX.L'L	Vector length/RC	P[22:21]	
EVEX.z	Zeroing/Merging	P[23]	

The bit fields in P[23:0] are divided into the following functional groups (Table 2-32 provides a tabular summary):

- Reserved bits: P[3] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD.
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- EVEX.mmm: P[2:0] provides access to up to eight decoding maps. Currently, only the following decoding maps are supported: 1, 2, 3, 5, and 6. Map ids 1, 2, and 3 are denoted by 0F, 0F38, and 0F3A, respectively, in the instruction encoding descriptions.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:

— Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.

— Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).

— Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.

— For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.

- Vector length/rounding control specifier: P[22:21] can serve one of three options.

  — Vector length information for packed vector instructions.

  — Ignored for instructions operating on vector register content as a single data element.

  — Rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

2.7.2 Register Specifier Encoding and EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 2-33. Opmask register encoding is described in Section 2.7.3.

Table 2-33. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits

	4[1]	3	[2:0]	Reg. Type	Common Usages
REG	EVEX.R'	REX.R	modrm.reg	GPR, Vector	Destination or Source
VVVV	EVEX.V'	EVEX.vvvv		GPR, Vector	2ndSource or Destination
RM	EVEX.X	EVEX.B	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	0	EVEX.B	modrm.r/m	GPR	memory addressing
INDEX	0	EVEX.X	sib.index	GPR	memory addressing
VIDX	EVEX.V'	EVEX.X	sib.index	Vector	VSIB memory addressing

NOTES:

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 2-34.

Table 2-34. EVEX Encoding Register Specifiers in 32-bit Mode

	[2:0]	Reg. Type	Common Usages
REG	modrm.reg	GPR, Vector	Destination or Source
VVVV	EVEX.vvv	GPR, Vector	2nd Source or Destination
RM	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	modrm.r/m	GPR	Memory Addressing
INDEX	sib.index	GPR	Memory Addressing
VIDX	sib.index	Vector	VSIB Memory Addressing

2.7.3 Opmask Register Encoding

There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.
- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 2.7.4).
- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

Table 2-35. Opmask Register Specifier Encoding

	[2:0]	Register Access	Common Usages
REG	modrm.reg	k0-k7	Source
VVVV	VEX.vvvv	k0-k7	2nd Source
RM	modrm.r/m	k0-7	1st Source
{k1}	EVEX.aaa	k0[1]-k7	Opmask

NOTES:

1. Instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.

2.7.4 Masking Support in EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.
- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided into the following groups:

- Instructions which support "zeroing-masking".
  - Also allow merging-masking.
- Instructions which require aaa = 000.
  - Do not allow any form of masking.
- Instructions which allow merging-masking but do not allow zeroing-masking.
  - Require EVEX.z to be set to 0.
  - This group is mostly composed of instructions that write to memory.
- Instructions which require aaa <> 000 do not allow EVEX.z to be set to 1.
  - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

2.7.5 Compressed Displacement (disp8*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-36 and Table 2-37 below, where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-36 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.7.11).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less then dword do not support broadcasting using EVEX.b. These are listed in Table 2-37. Table 2-37 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-37. Instruction classified in Table 2-37 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8*N rules still apply when using 16b addressing.

Table 2-36. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast

TupleType	EVEX.b	InputSize	EVEX.W	Broadcast	N (VL=128)	N (VL=256)	N (VL= 512)	Comment
Full	0	32bit	0	none	16	32	64	Load+Op (Full Vector Dword/Qword)
	1	32bit	0	{1tox}	4	4	4	
	0	64bit	1	none	16	32	64	
	1	64bit	1	{1tox}	8	8	8	
Half	0	32bit	0	none	8	16	32	Load+Op (Half Vector)
	1	32bit	0	{1tox}	4	4	4	

Table 2-37. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Full Mem	N/A	N/A	16	32	64	Load/store or subDword full vector
Tuple1 Scalar	8bit	N/A	1	1	1	1Tuple
	16bit	N/A	2	2	2	
	32bit	0	4	4	4	
	64bit	1	8	8	8	
Tuple1 Fixed	32bit	N/A	4	4	4	1 Tuple, memsize not affected by EVEX.W
	64bit	N/A	8	8	8	
Tuple2	32bit	0	8	8	8	Broadcast (2 elements)
	64bit	1	NA	16	16	
Tuple4	32bit	0	NA	16	16	Broadcast (4 elements)
	64bit	1	NA	NA	32	
Tuple8	32bit	0	NA	NA	32	Broadcast (8 elements)
Half Mem	N/A	N/A	8	16	32	SubQword Conversion
Quarter Mem	N/A	N/A	4	8	16	SubDword Conversion

Table 2-37. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast (Contd.)

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Eighth Mem	N/A	N/A	2	4	8	SubWord Conversion
Mem128	N/A	N/A	16	16	16	Shift count from memory
MOVDDUP	N/A	N/A	8	32	64	VMOVDDUP

2.7.6 EVEX Encoding of Broadcast/Rounding/SAE Support

EVEX.b can provide three types of encoding context, depending on the instruction classes:

- Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with "load+op" semantic.
- Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.
- "Suppress All exceptions" (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that do not have rounding semantic.

2.7.7 Embedded Broadcast Support in EVEX

EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

- Instructions with only one scalar result is written to the vector destination.
- Instructions with explicit broadcast functionality provided by its opcode.
- Instruction semantic is a pure load or a pure store operation.

2.7.8 Static Rounding Support in EVEX

Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L'L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, "suppress all exceptions" is implied. The processor behaves as if all MXCSR masking controls are set.

2.7.9 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, "suppress all exceptions" is implied. The processor behaves as if all MXCSR masking controls are set.

2.7.10 Vector Length Orthogonality

The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L'L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L'L bits. When EVEX is used to encode scalar instructions, L'L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 2-38.

Table 2-38. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions

Position	P2[4]	P2[6:5]	P2[6:5]
Broadcast/Rounding/SAE Context	**EVEX.b**	**EVEX.L'L**	**EVEX.RC**
Reg-reg, FP Instructions w/ rounding semantic or SAE	Enable static rounding control (SAE implied)	Vector length Implied (512 bit or scalar)	00b: SAE + RNE 01b: SAE + RD 10b: SAE + RU 11b: SAE + RZ
Load+op Instructions w/ memory source	Broadcast Control	00b: 128-bit 01b: 256-bit 10b: 512-bit 11b: Reserved (#UD)	NA
Other Instructions (Explicit Load/Store/Broadcast/Gather/Scatter)	Must be 0 (otherwise #UD)		NA

2.7.11 #UD Equations for EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

2.7.11.1 State Dependent #UD

In general, attempts of execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-39 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-39 will cause #UD.

Table 2-39. OS XSAVE Enabling Requirements of Instruction Categories

Instruction Categories	Vector Register State Access	Required XCR0 Bit Vector [7:0]
Legacy SIMD prefix encoded Instructions (e.g SSE)	XMM	xxxxxx11b
VEX-encoded instructions operating on YMM	YMM	xxxxx111b
EVEX-encoded 128-bit instructions	ZMM	111xx111b
EVEX-encoded 256-bit instructions	ZMM	111xx111b
EVEX-encoded 512-bit instructions	ZMM	111xx111b
VEX-encoded instructions operating on opmask	k-reg	111xxx11b

2.7.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-40.

Table 2-40. Opcode Independent, State Dependent EVEX Bit Fields[1]

Position	Notation	64-bit #UD	Non-64-bit #UD
P[3]	--	if > 0	if > 0
P[10]	--	if 0	if 0
P[2:0]	EVEX.mmm	if 000b, 100b, or 111b	if 000b, 100b, or 111b
P[7 : 6]	EVEX.RX	None (valid)	None (BOUND if EVEX.RX != 11b)

NOTES:

1. This table is also representative of VEX restrictions. For VEX operations, use the Notation field.

2.7.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-41 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

Table 2-41. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields[1]

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.R	P[7]	ModRM.reg encodes k-reg	If EVEX.R = 0	None (BOUND if EVEX.RX != 11b)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes all other registers	None (valid)	
EVEX.X	P[6]	ModRM.r/m encodes ZMM/YMM/XMM	None (valid)	
		ModRM.r/m encodes k-reg or GPR	None (ignored)	
		ModRM.r/m without SIB/VSIB	None (ignored)	
		ModRM.r/m with SIB/VSIB	None (valid)	
EVEX.B	P[5]	ModRM.r/m encodes k-reg	None (ignored)	None (ignored)
		ModRM.r/m encodes other registers	None (valid)	
		ModRM.r/m base present	None (valid)	
		ModRM.r/m base not present	None (ignored)	
EVEX.R'	P[4]	ModRM.reg encodes k-reg or GPR	If 0	None (ignored)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes ZMM/YMM/XMM	None (valid)	
EVEX.vvvv	P[14:11]	vvvv encodes ZMM/YMM/XMM	None (valid)	None (valid) P[14] ignored
		Otherwise	If != 1111b	If != 1111b
EVEX.V'	P[19]	Encodes ZMM/YMM/XMM	None (valid)	If 0
		Otherwise	If 0	If 0

NOTES:

1. This table also represents VEX restrictions.

Table 2-42 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

Table 2-42. #UD Conditions of Opmask Related Encoding Field

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.aaa	P[18:16]	Instructions do not use opmask for conditional processing[1].	If aaa != 000b	If aaa != 000b
		Opmask used as conditional processing mask and updated at completion[2].	If aaa = 000b	If aaa = 000b;
		Opmask used as conditional processing.	None (valid[3])	None (valid[1])
EVEX.z	P[23]	Vector instruction using opmask as source or destination[4].	If EVEX.z != 0	If EVEX.z != 0
		Store instructions or gather/scatter instructions.	If EVEX.z != 0	If EVEX.z != 0
		Instructions with EVEX.aaa = 000b.	If EVEX.z != 0	If EVEX.z != 0
VEX.vvvv	Varies	K-regs are instruction operands not mask control.	If vvvv = 0xxxb	None

NOTES:

1. E.g., VPBROADCASTMxxx, VPMOVM2x, VPMOVx2M.
2. E.g., Gather/Scatter family.

3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of 'all ones' regardless of the actual value in K0.
4. E.g., VFPCLASSPD/PS, VCMPB/D/Q/W family, VPMOVM2x, VPMOVx2M.

Table 2-43 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

Table 2-43. #UD Conditions Dependent on EVEX.b Context

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.L'Lb	P[22 : 20]	Reg-reg, FP instructions with rounding semantic.	None (valid[1])	None (valid[1])
		Other reg-reg, FP instructions that can cause #XM.	None (valid[2])	None (valid[2])
		Other reg-mem instructions in Table 2-36.	None (valid[3])	None (valid[3])
		Other instruction classes[4] in Table 2-37.	If EVEX.b = 1	If EVEX.b = 1

NOTES:

1. L'L specifies rounding control, see Table 2-38, supports {er} syntax.
2. L'L is ignored.
3. L'L specifies vector length, see Table 2-38, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

2.7.12 Device Not Available

EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3]= 1.

2.7.13 Scalar Instructions

EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

2.8 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of "E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

Table 2-44. EVEX-Encoded Instruction Exception Class Summary

Exception Class	Instruction set	Mem arg	(#XM)
Type E1	Vector Moves/Load/Stores	Explicitly aligned, w/ fault suppression	None
Type E1NF	Vector Non-temporal Stores	Explicitly aligned, no fault suppression	None
Type E2	FP Vector Load+op	Support fault suppression	Yes
Type E2NF	FP Vector Load+op	No fault suppression	Yes
Type E3	FP Scalar/Partial Vector, Load+Op	Support fault suppression	Yes
Type E3NF	FP Scalar/Partial Vector, Load+Op	No fault suppression	Yes
Type E4	Integer Vector Load+op	Support fault suppression	No
Type E4NF	Integer Vector Load+op	No fault suppression	No
Type E5	Legacy-like Promotion	Varies, Support fault suppression	No
Type E5NF	Legacy-like Promotion	Varies, No fault suppression	No
Type E6	Post AVX Promotion	Varies, w/ fault suppression	No
Type E6NF	Post AVX Promotion	Varies, no fault suppression	No
Type E7NM	Register-to-register op	None	None
Type E9NF	Miscellaneous 128-bit	Vector-length Specific, no fault suppression	None
Type E10	Non-XF Scalar	Vector Length ignored, w/ fault suppression	None
Type E10NF	Non-XF Scalar	Vector Length ignored, no fault suppression	None
Type E11	VCVTPH2PS, VCVTPS2PH	Half Vector Length, w/ fault suppression	Yes
Type E12	Gather and Scatter Family	VSIB addressing, w/ fault suppression	None
Type E12NP	Gather and Scatter Prefetch Family	VSIB addressing, w/o page fault	None

Table 2-45 lists EVEX-encoded instruction mnemonic by exception classes.

Table 2-45. EVEX Instructions in Each Exception Class

Exception Class	Instruction
Type E1	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
Type E1NF	VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
Type E2	VADDPD, VADDPH, VADDPS, VCMPPD, VCMPPH, VCMPPS, VCVTDQ2PH, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PH, VCVTPD2PS, VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPH2DQ, VCVTPH2PD, VCVTPH2QQ, VCVTPH2UDQ, VCVTPH2UQQ, VCVTPH2UW, VCVTPH2W, VCVTPS2DQ, VCVTPS2PD, VCVTPS2QQ, VCVTPS2UDQS, VCVTPS2UQQ, VCVTQQ2PD, VCVTQQ2PH, VCVTQQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPH2DQ, VCVTTPH2QQ, VCVTTPH2UDQ, VCVTTPH2UQQ, VCVTTPH2UW, VCVTTPH2W, VCVTTPS2DQ, VCVTTPS2QQ, VCVTTPS2UDQ, VCVTTPS2UQQ, VCVTUDQ2PH, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PH, VCVTUQQ2PS, VCVTUW2PH, VCVTW2PH, VDIVPD, VDIVPH, VDIVPS, VEXP2PD, VEXP2PS, VFIXUPIMMPD, VFIXUPIMMPS, VFMADDxxxPD, VFMADDxxxPH, VFMADDxxxPS, VFMADDSUBxxxPD, VFMADDSUBxxxPH, VFMADDSUBxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPH, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPH, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPH, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPH, VFNMSUBxxxPS, VGETEXPPD, VGETEXPPH, VGETEXPPS, VGETMANTPD, VGETMANTPH, VGETMANTPS, VGETMANTSH, VMAXPD, VMAXPH, VMAXPS, VMINPD, VMINPH, VMINPS, VMULPD, VMULPH, VMULPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPH, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPH, VRNDSCALEPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS, VSCALEFPD, VSCALEFPS, VSQRTPD, VSQRTPH, VSQRTPS, VSUBPD, VSUBPH, VSUBPS

Table 2-45. EVEX Instructions in Each Exception Class (Contd.)

Exception Class	Instruction
Type E3	VADDSD, VADDSH, VADDSS, VCMPSD, VCMPSH, VCMPSS, VCVTSD2SH, VCVTSD2SS, VCVTSH2SD, VCVTSH2SS, VCVTSS2SD, VCVTSS2SH, VDIVSD, VDIVSH, VDIVSS, VFMADDxxxSD, VFMADDxxxSH, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSH, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSH, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSH, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSH, VGETEXPSS, VGETMANTSD, VGETMANTSH, VGETMANTSS, VMAXSD, VMAXSH, VMAXSS, VMINSD, VMINSH, VMINSS, VMULSD, VMULSH, VMULSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESH, VREDUCESS, VRNDSCALESD, VRNDSCALESH, VRNDSCALESS, VSCALEFSD, VSCALEFSH, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS, VSQRTSD, VSQRTSH, VSQRTSS, VSUBSD, VSUBSH, VSUBSS
Type E3NF	VCOMISD, VCOMISH, VCOMISS, VCVTSD2SI, VCVTSD2USI, VCVTSH2SI, VCVTSH2USI, VCVTSI2SD, VCVTSI2SH, VCVTSI2SS, VCVTSS2SI, VCVTSS2USI, VCVTTSD2SI, VCVTTSD2USI, VCVTTSH2SI, VCVTTSH2USI, VCVTTSS2SI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SH, VCVTUSI2SS, VUCOMISD, VUCOMISH, VUCOMISS
Type E4	VANDPD, VANDPS, VANDNPD, VANDNPS, VBLENDMPD, VBLENDMPS, VFCMADDCPH, VFCMULCPH, VFMADDCPH, VFMULCPH, VFPCLASSPD, VFPCLASSPH, VFPCLASSPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDD, VPADDQ, VPANDD, VPANDQ, VPANDND, VPANDNQ, VPBLENDMB, VPBLENDMD, VPBLENDMQ, VPBLENDMW, VPCMPD, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPMADD52LUQ, VPMADD52HUQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPROLD, VPROLQ, VPROLVD, VPROLVQ, VPRORD, VPRORQ, VPRORVD, VPRORVQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRAVW, VPSRAVD, VPSRAVW, VPSRAVQ, VPSRLD, VPSRLQ)[1], VPSUBD, VPSUBQ, VPSUBUSB, VPSUBUSW, VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VPXORD, VPXORQ, VPSLLVD, VPSLLVQ, VRCP14PD, VRCP14PS, VRCPPH, VRSQRT14PD, VRSQRT14PS, VRSQRTPH, VXORPD, VXORPS
E4.nb[2]	VCOMPRESSPD, VCOMPRESSPS, VEXPANDPD, VEXPANDPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VMOVUPD, VMOVUPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPCMPB, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPCMPW, VPCMPUB, VPCMPUW, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW, VPMULLW, VPSLLVW, VPSLLW, VPSRAW, VPSRLVW, VPSRLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW
Type E4NF	VALIGND, VALIGNQ, VPACKSSDW, VPACKUSDW, VPCONFLICTD, VPCONFLICTQ, VPERMD, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMPD, VPERMPS, VPERMQ, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VPMULTISHIFTQB, VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2, VSHUFPD, VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS
E4NF.nb[2]	VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP, VPSADBW, VPSHUFB, VPSHUFHW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)[3], VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD, VPERMW, VPERMI2W, VPERMT2W
Type E5	PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBW, PMOVZXBD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ, VCVTDQ2PD, VCVTUDQ2PD, VMOVSH, VPMOVSXxx, VPMOVZXxx,
Type E5NF	VMOVDDUP
Type E6	VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VBROADCASTSD, VBROADCASTSS, VFPCLASSSD, VFPCLASSSS, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW, VPMOVWB, VPMOVSWB, VPMOVUSWB
Type E6NF	VEXTRACTF32X4, VEXTRACTF32X8, VEXTRACTF64X2, VEXTRACTF64X4, VEXTRACTI32X4, VEXTRACTI32X8, VEXTRACTI64X2, VEXTRACTI64X4, VINSERTF32X4, VINSERTF32X8, VINSERTF64X2, VINSERTF64X4, VINSERTI32X4, VINSERTI32X8, VINSERTI64X2, VINSERTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D

Table 2-45. EVEX Instructions in Each Exception Class (Contd.)

Exception Class	Instruction
Type E7NM.128[4]	VMOVHLPS, VMOVLHPS
Type E7NM.	(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW)[5], VPMOVB2M, VPMOVD2M, VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVQ2M, VPMOVW2M
Type E9NF	VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VMOVW, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ
Type E10	VFCMADDCSH, VFMADDCSH, VFCMULCSH, VFMULCSH, VFPCLASSSH, VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRCPSH, VRSQRT14SD, VRSQRT14SS, VRSQRTSH
Type E10NF	(VCVTSI2SD, VCVTUSI2SD)[6]
Type E11	VCVTPH2PS, VCVTPS2PH
Type E12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS
Type E12NP	VGATHERPF0DPD, VGATHERPF0DPS, VGATHERPF0QPD, VGATHERPF0QPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPF0DPD, VSCATTERPF0DPS, VSCATTERPF0QPD, VSCATTERPF0QPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS

NOTES:

1. Operand encoding Full tupletype with immediate.
2. Embedded broadcast is not supported with the ".nb" suffix.
3. Operand encoding Mem128 tupletype.
4. #UD raised if EVEX.L'L !=00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.

2.8.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1.

Table 2-46. Type E1 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.

EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF.

Table 2-47. Type E1NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.8.2 Exceptions Type E2 of EVEX-Encoded Instructions

EVEX-encoded vector instructions with arithmetic semantic follow exception class E2.

Table 2-48. Type E2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.8.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

EVEX-encoded scalar instructions with arithmetic semantic that support memory fault suppression follow exception class E3.

Table 2-49. Type E3 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSX-MMEXCPT[bit 10] = 1.

EVEX-encoded scalar instructions with arithmetic semantic that do not support memory fault suppression follow exception class E3NF.

Table 2-50. Type E3NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			EVEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.8.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E4.

Table 2-51. Type E4 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43 and in E4.nb subclass (see E4.nb entries in Table 2-45). ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E4NF.

Table 2-52. Type E4NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-45). ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.8.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

Table 2-53. Type E5 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.

Table 2-54. Type E5NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.6 Exceptions Type E6 and E6NF

Table 2-55. Type E6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

Table 2-56. Type E6NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.7 Exceptions Type E7NM

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM.

Table 2-57. Type E7NM Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.8.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

Table 2-58. Type E9 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

Table 2-59. Type E9NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.9 Exceptions Type E10 and E10NF

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, do not cause a SIMD FP exception, and support memory fault suppression follow exception class E10.

Table 2-60. Type E10 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, do not cause a SIMD FP exception, and do not support memory fault suppression follow exception class E10NF.

Table 2-61. Type E10NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.8.10 Exceptions Type E11 (EVEX-only, Mem Arg, No AC, Floating-point Exceptions)

EVEX-encoded instructions that can cause SIMD FP exception, memory operand support fault suppression but do not cause #AC follow exception class E11.

Table 2-62. Type E11 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1.

2.8.11 Exceptions Type E12 and E12NP (VSIB Mem Arg, No AC, No Floating-point Exceptions)

Table 2-63. Type E12 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met. ▪ If vvvv != 1111b.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
	X	X	X	X	If index = destination register (gather operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

Table 2-64. Type E12NP Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41. ▪ Opmask encoding #UD condition of Table 2-42. ▪ EVEX.b encoding #UD condition of Table 2-43. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

2.9 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS, TYPE K20 AND TYPE K21

The exception behavior of VEX-encoded opmask instructions are listed below.

2.9.1 Exceptions Type K20

Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

Table 2-65. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If ModRM:[7:6] != 11b.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

2.9.2 Exceptions Type K21

Exception conditions of Opmask instructions that address memory are listed as Type K21.

Table 2-66. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-39 not met. ▪ Opcode independent #UD condition in Table 2-40. ▪ Operand encoding #UD conditions in Table 2-41.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, #SS(0)	X	X	X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.

2.10 INTEL® AMX INSTRUCTION EXCEPTION CLASSES

Alignment exceptions: The Intel AMX instructions that access memory will never generate #AC exceptions.

Table 2-67. Intel® AMX Exception Classes

Class	Description
AMX-E1	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.
	• #GP based on palette and configuration checks (see pseudocode). • #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if a page fault occurs.
AMX-E2	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.
	• #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if a page fault occurs.
AMX-E3	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111. • #UD if not using SIB addressing. • #UD if TILES_CONFIGURED == 0. • #UD if tsrc or tdest are not valid tiles. • #UD if tsrc/tdest are ≥ palette_table[tilecfg.palette_id].max_names. • #UD if tsrc.colbytes mod 4 ≠ 0 OR tdest.colbytes mod 4 ≠ 0. • #UD if tilecfg.start_row ≥ tsrc.rows OR tilecfg.start_row ≥ tdest.rows.
	• #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if any memory operand causes a page fault.
	• #NM if XFD[18] == 1.

Table 2-67. Intel® AMX Exception Classes (Contd.)

Class	Description
AMX-E4	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if srcdest == src1 OR src1 == src2 OR srcdest == src2. • #UD if TILES_CONFIGURED == 0. • #UD if srcdest.colbytes mod 4 ≠ 0. • #UD if src1.colbytes mod 4 ≠ 0. • #UD if src2.colbytes mod 4 ≠ 0. • #UD if srcdest/src1/src2 are not valid tiles. • #UD if srcdest/src1/src2 are ≥ palette_table[tilecfg.palette_id].max_names. • #UD if srcdest.colbytes ≠ src2.colbytes. • #UD if srcdest.rows ≠ src1.rows. • #UD if src1.colbytes / 4 ≠ src2.rows. • #UD if srcdest.colbytes > tmul_maxn. • #UD if src2.colbytes > tmul_maxn. • #UD if src1.colbytes/4 > tmul_maxk. • #UD if src2.rows > tmul_maxk.
	• #NM if XFD[18] == 1.
AMX-E5	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111. • #UD if TILES_CONFIGURED == 0. • #UD if tdest is not a valid tile. • #UD if tdest is ≥ palette_table[tilecfg.palette_id].max_names.
	• #NM if XFD[18] == 1.
AMX-E6	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.

6. Updates to Chapter 3, Volume 2A

Change bars and violet text show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A:* Instruction Set Reference, A-L.

Changes to this chapter:

- Revised opcode tables removing REX+ prefixes for instructions: ADC, ADD, AND, CMP, CMPXCHG, CRC32, DEC, DIV, IDIV, IMUL, INC, LDS/LES/LFS/LGS/LSS.
- Updated the Operation Flags Affected sections for BSF and BSR instructions.
- Added explanation of REX.R format in the Description of the BTC instruction.
- Removed incorrect cross-reference in CMPPD, CMPPS, CMPSD, and CMPSS.
- Updated CPUID Leaf 24H bits 18:16 to Reserved at 111.
- Revised Description to provide return integer value for convert instructions: CVTPD2DQ, CVTPD2PI, CVTPS2DQ, CVTPS2PI, CVTSD2SI, CVTSS2SI, CVTTPD2DQ, CVTTPD2PI, CVTTPS2DQ, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI.
- Corrected the exception type for the EVEX-encoded instruction of CVTPS2PD (VCVTPS2PD).
- For the LAR and LSL instructions, revised the opcode table and added clarification in the Description for 16-bit, 32-bit, and 64-bit operand sizes. For the LAR instruction, removed "Byte" from the title. For the LSL instruction, added content for CF, OF, SF, AF, and PF flags.
- Revised Description of LZCNT instruction.
- Removed footnote references to verify vector options for the following instructions:
  - ADDPD
  - ADDPS
  - ADDSD
  - ADDSS
  - AESDEC
  - AESDECLAST
  - AESENC
  - AESENCLAST
  - ANDNPD
  - ANDNPS
  - ANDPD
  - ANDPS
  - CMPPD
  - CMPPS
  - CMPSD
  - CMPSS
  - COMISD
  - COMISS
  - CVTDQ2PD
  - CVTDQ2PS
  - CVTPD2DQ
  - CVTPD2PS
  - CVTPS2DQ
  - CVTPS2PD
  - CVTSD2SI
  - CVTSD2SS

— CVTSI2SD
— CVTSI2SS
— CVTSS2SD
— CVTSS2SI
— CVTTPD2DQ
— CVTTSD2SI
— CVTTSS2SI
— DIVPD
— DIVPS
— DIVSD
— DIVSS
— EXTRACTPS
— GF2P8AFFINEINVQB
— GF2P8AFFINEQB
— GF2P8MULB
— INSERTPS

CHAPTER 3
INSTRUCTION SET REFERENCE, A-L

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-L) in IA-32e, protected, virtual-8086, and real-address modes of operation. The set includes general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3/SSSE3/SSE4, AESNI/PCLMULQDQ, AVX, and system instructions. See also Chapter 4, "Instruction Set Reference, M-U," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B; Chapter 5, "Instruction Set Reference, V," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C; and Chapter 6, "Instruction Set Reference, W-Z," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections.

3.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The heading below introduces the example. The table below provides an example summary table.

CMC—Complement Carry Flag [this is an example]

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F5	CMC	ZO	V/V	N/A	Complement carry flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix)

The "Opcode" column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **NP** — Indicates the use of 66/F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- **NFx** — Indicates the use of F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r —** Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **cb, cw, cd, cp, co, ct —** A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id, io —** A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords, and quadwords are given with the low-order byte first.
- **+rb, +rw, +rd, +ro —** Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. "+ro" is applicable only in 64-bit mode. See Table 3-1 for the codes.
- +**i —** A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BL	None	3	BX	None	3	EBX	None	3	RBX	None	3
AH	Not encodable (N.E.)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A
CH	N.E.	5	BP	None	5	EBP	None	5	N/A	N/A	N/A
DH	N.E.	6	SI	None	6	ESI	None	6	N/A	N/A	N/A
BH	N.E.	7	DI	None	7	EDI	None	7	N/A	N/A	N/A
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro (Contd.)

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
SIL	Yes	6	SI	None	6	ESI	None	6	RSI	None	6
DIL	Yes	7	DI	None	7	EDI	None	7	RDI	None	7
Registers R8 - R15 (see below): Available in 64-Bit Mode Only											
R8B	Yes	0	R8W	Yes	0	R8D	Yes	0	R8	Yes	0
R9B	Yes	1	R9W	Yes	1	R9D	Yes	1	R9	Yes	1
R10B	Yes	2	R10W	Yes	2	R10D	Yes	2	R10	Yes	2
R11B	Yes	3	R11W	Yes	3	R11D	Yes	3	R11	Yes	3
R12B	Yes	4	R12W	Yes	4	R12D	Yes	4	R12	Yes	4
R13B	Yes	5	R13W	Yes	5	R13D	Yes	5	R13	Yes	5
R14B	Yes	6	R14W	Yes	6	R14D	Yes	6	R14	Yes	6
R15B	Yes	7	R15W	Yes	7	R15D	Yes	7	R15	Yes	7

3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix)

In the Instruction Summary Table, the Opcode column presents each instruction encoded using the VEX prefix in following form (including the modR/M byte if applicable, the immediate byte if applicable):

VEX.[128,256].[66,F2,F3].0F/0F3A/0F38.[W0,W1] opcode [/r] [/ib,/is4]

- **VEX** — Indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 2.3 for more detail on the VEX prefix.

  The encoding of various sub-fields of the VEX prefix is described using the following notations:

  — **128,256:** VEX.L field can be 0 (denoted by VEX.128, VEX.L0, or VEX.LZ) or 1 (denoted by VEX.256 or VEX.L1). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.256 or VEX.128 in the opcode column should be interpreted as follows:

    - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
    - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g., VMOVLPS).
    - If VEX.L0 or VEX.L1 is present in the opcode column: The specified VEX.L value is required for encoding this instruction but does not have the connotation of specifying vector length.
    - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either “SS“, “SD”, or “SI” for SIMD floating-point conversion instructions.
    - If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.

— **66,F2,F3:** The presence or absence of these values map to the VEX.pp field encodings. If absent, this corresponds to VEX.pp=00B. If present, the corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix. The VEX.pp field may be encoded using either the 2-byte or 3-byte form of the VEX prefix.

— **0F,0F3A,0F38:** The presence maps to a valid encoding of the VEX.mmmmm field. Only three encoded values of VEX.mmmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH, and 0F38H. The effect of a valid VEX.mmmmm encoding on the ensuing opcode byte is same as if the corresponding escape byte sequence on the ensuing opcode byte for non-VEX encoded instructions. Thus a valid encoding of VEX.mmmmm may be consider as an implies escape byte sequence of either 0FH, 0F3AH or 0F38H. The VEX.mmmmm field must be encoded using the 3-byte form of VEX prefix.

— **0F,0F3A,0F38 and 2-byte/3-byte VEX:** The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded by the two-byte of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.

— **W0:** VEX.W=0.

— **W1:** VEX.W=1.

— The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 2.3 on the subfield definitions within VEX.

— **WIG:** can use C5H form (if not requiring VEX.mmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.

— If WIG is present, the instruction may be encoded using either the two-byte form or the three-byte form of VEX. When encoding the instruction using the three-byte form of VEX, the value of VEX.W is ignored.

- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in either imm8[7:4] (for 64-bit mode) or imm8[6:4] (for 32-bit mode), and instruction-specific payload in imm8[3:0].
- In general, the encoding o f VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

EVEX.[128,256,512,LLIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [/ib]

- **EVEX** — The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 2.7.1 for more detail on the EVEX prefix.

  The encoding of various sub-fields of the EVEX prefix is described using the following notations:

  — **128, 256, 512, LLIG:** This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.

  — **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.

  — **0F,0F3A,0F38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH, and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.

  — **W0:** EVEX.W=0.

— **W1:** EVEX.W=1.

— **WIG:** EVEX.W bit ignored

- **opcode** — Instruction opcode.
- In general, the encoding of EVEX.R and R’, EVEX.X and X’, and EVEX.B and B’ fields are not shown explicitly in the opcode column.

NOTE

Previously, the terms NDS, NDD, and DDS were used in instructions with an EVEX (or VEX) prefix. These terms indicated that the vvvv field was valid for encoding, and specified register usage. These terms are no longer necessary and are redundant with the instruction operand encoding tables provided with each instruction. The instruction operand encoding tables give explicit details on all operands, indicating where every operand is stored and if they are read or written. If vvvv is not listed as an operand in the instruction operand encoding table, then EVEX (or VEX) vvvv must be 0b1111.

3.1.1.3 Instruction Column in the Opcode Summary Table

The “Instruction” column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16, rel32** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
- **ptr16:16, ptr16:32** — A far pointer, typically to a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL, and SIL; or one of the byte registers (R8B - R15B) available when using REX.R and 64-bit mode.
- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.
- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.
- **r64** — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15. These are available when using REX.R and 64-bit mode.
- **imm8** — An immediate byte value. The imm8 symbol can be a signed number between –128 and +127 inclusive; an unsigned number between 0 and 255 inclusive; or a bitmap when an instruction uses its individual bits. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.
- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.
- **imm64** — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and –9,223,372,036,854,775,808 inclusive.
- **/ib** — A single-byte value.

- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL, and SIL) or a byte from memory. Byte registers R8B - R15B are available using REX.R in 64-bit mode.
- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.
- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.
- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **reg** — A general-purpose register used for instructions when the width of the register does not matter to the semantics of the operation of the instruction. The register can be r16, r32, or r64.
- **m** — A 16-, 32- or 64-bit operand in memory.
- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.
- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** — A doubleword operand in memory. The contents of memory are found at the address provided by the effective address computation.
- **m64** — A memory quadword operand in memory.
- **m128** — A memory double quadword operand in memory.
- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.
- **m80bcd**— A Binary Coded Decimal (BCD) operand in memory, 80 bits.
- **moffs8, moffs16, moffs32, moffs64** — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.
- **Sreg** — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.
- **m32fp, m64fp, m80fp** — A single precision, double precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.
- **m16int, m32int, m64int** — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.
- **ST or ST(0)** — The top element of the FPU register stack.
- **ST(i)** — The i^{th} element from the top of the FPU register stack (*i* := 0 through 7).
- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.

- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.
- **xmm/m32**— An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **<XMM0>**— Indicates implied use of the XMM0 register.

  When there is ambiguity, xmm1 indicates the first source operand using an XMM register and xmm2 the second source operand using an XMM register.

  Some instructions use the XMM0 register as the third source operand, indicated by <XMM0>. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.
- **ymm** — A YMM register. The 256-bit YMM registers are: YMM0 through YMM7**;** YMM8 through YMM15 are available in 64-bit mode.
- **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX instructions.
- **ymm/m256** — A YMM register or 256-bit memory operand.
- **<YMM0>**— Indicates use of the YMM0 register as an implicit argument.
- **bnd** — A 128-bit bounds register. BND0 through BND3.
- **mib** — A memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.
- **m512** — A 64-byte operand in memory.
- **zmm/m512** — A ZMM register or 512-bit memory operand.
- **{k1}{z}** — A mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.
- **{k1}** — Without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.
- **k1** — A mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.
- **mV** — A vector memory operand; the operand size is dependent on the instruction.
- **vm32{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).
- **vm64{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).

- **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.
- **<ZMM0>** — Indicates use of the ZMM0 register as an implicit argument.
- **{er}** — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.
- **SRC** — The source in a single-source instruction.
- **DST** — The destination in an instruction. This field is encoded by reg_field.

In the instruction encoding, the MODRM byte is represented several ways depending on the role it plays. The MODRM byte has 3 fields: 2-bit MODRM.MOD field, a 3-bit MODRM.REG field and a 3-bit MODRM.RM field. When all bits of the MODRM byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the MODRM byte must contain fixed values, those values are specified as follows:

- If only the MODRM.MOD must be 0b11, and MODRM.REG and MODRM.RM fields are unrestricted, this is denoted as **11:rrr:bbb**. The **rrr** correspond to the 3-bits of the MODRM.REG field and the **bbb** correspond to the 3-bits of the MODMR.RM field.
- If the MODRM.MOD field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then we use the notation !(11).
- If the MODRM.REG field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

3.1.1.4 Operand Encoding Column in the Instruction Summary Table

The "operand encoding" column is abbreviated as Op/En in the Instruction Summary table heading. Instruction operand encoding information is provided for each assembly instruction syntax using a letter to cross reference to a row entry in the operand encoding definition table that follows the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

EVEX encoded instructions employ compressed disp8*N encoding of the displacement bytes, where N is defined in Table 2-36 and Table 2-37, according to tupletypes. The tupletype for an instruction is listed in the operand encoding definition table where applicable.

NOTES

- The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.
- In the encoding definition table, the letter 'r' within a pair of parenthesis denotes the content of the operand will be read by the processor. The letter 'w' within a pair of parenthesis denotes the content of the operand will be updated by the processor.

3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table

The "64/32-bit Mode" column indicates whether the opcode sequence is supported in (a) 64-bit mode or (b) the Compatibility mode and other IA-32 modes that apply in conjunction with the CPUID feature flag associated specific instruction extensions.

The 64-bit mode support is to the left of the 'slash' and has the following notation:

- **V** — Supported.
- **I** — Not supported.
- **N.E.** — Indicates an instruction syntax is not encodable in 64-bit mode (it may represent part of a sequence of valid instructions in other modes).
- **N.P.** — Indicates the REX prefix does not affect the legacy instruction in 64-bit mode.
- **N.I.** — Indicates the opcode is treated as a new instruction in 64-bit mode.
- **N.S.** — Indicates an instruction syntax that requires an address override prefix in 64-bit mode and is not supported. Using an address override prefix in 64-bit mode may result in model-specific execution behavior.

The Compatibility/Legacy Mode support is to the right of the 'slash' and has the following notation:

• **V —** Supported.
• **I —** Not supported.
• **N.E. —** Indicates an Intel 64 instruction mnemonics/syntax that is not encodable; the opcode sequence is not applicable as an individual instruction in compatibility mode or IA-32 mode. The opcode may represent a valid sequence of legacy IA-32 instructions.

3.1.1.6 CPUID Support Column in the Instruction Summary Table

The fourth column holds abbreviated CPUID feature flags (e.g., appropriate bit in CPUID.01H.ECX, CPUID.01H.EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AESNI/PCLMULQDQ/AVX/RDRAND support) that indicate processor support for the instruction. If the corresponding flag is '0', the instruction will #UD.

3.1.1.7 Description Column in the Instruction Summary Table

The "Description" column briefly explains forms of the instruction.

3.1.1.8 Description Section

Each instruction is then described by number of information sections. The "Description" section describes the purpose of the instructions and required operands in more detail.

Summary of terms that may be used in the description section:

- **Legacy SSE** — Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ, and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
- **VEX.vvvv** — The VEX bit field specifying a source or destination register (in 1's complement form).
- **rm_field** — shorthand for the ModR/M *r/m* field and any REX.B.
- **reg_field** — shorthand for the ModR/M *reg* field and any REX.R.

3.1.1.9 Operation Section

The "Operation" section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE, and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.
- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32.

Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.

- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- A := B indicates that the value of B is assigned to A.
- The symbols =, ≠, >, <, ≥, and ≤ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.
- The expression "« COUNT" and "» COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** — The OperandSize identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The AddressSize identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

```
IF Instruction = MOVW
    THEN OperandSize := 16;
ELSE
    IF Instruction = MOVD
        THEN OperandSize := 32;
    ELSE
        IF Instruction = MOVQ
            THEN OperandSize := 64;
        FI;
    FI;
FI;
```

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for guidelines on how these attributes are determined.
- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
- **SRC** — Represents the source operand.
- **DEST** — Represents the destination operand.
- **MAXVL** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's encoding but is instead determined by the current value of XCR0. For details, refer to the table below. Note that the value of MAXVL is the largest of the features enabled. Future processors may define new bits in XCR0 whose setting may imply other values for MAXVL.

MAXVL Definition

XCR0 Component	MAXVL
XCR0.SSE	128
XCR0.AVX	256
XCR0.{ZMM_Hi256, Hi16_ZMM, OPMASK}	512

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value −10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than −128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than −32768, it is represented by the saturated value −32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than −128, it is represented by the saturated value −128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than −32768, it is represented by the saturated value −32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **LowOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the "Operation" subsection of the "PUSH—Push Word, Doubleword, or Quadword Onto the Stack" section in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.
- **Pop()** — removes the value from the top of the stack and returns it. The statement EAX := Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the "Operation" subsection in the "POP—Pop a Value From the Stack" section of Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.
- **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** — Performs a task switch.
- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

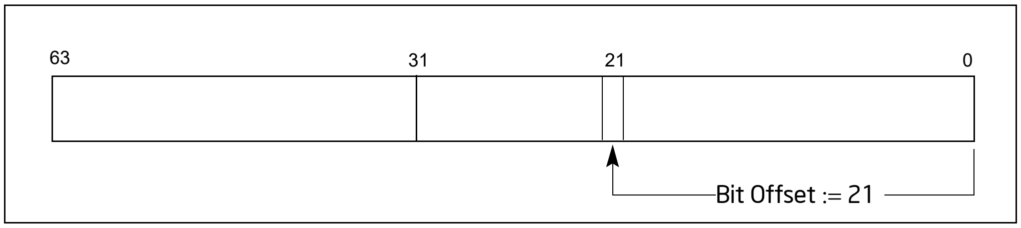


Figure 3-1. Bit Offset for BIT[RAX, 21]

If BitBase is a memory address, the BitOffset has different ranges depending on the operand size (see Table 3-2).

Table 3-2. Range of Bit Positions Specified by Bit Offset Operands

Operand Size	Immediate BitOffset	Register BitOffset
16	0 to 15	-2^{15} to $2^{15} - 1$
32	0 to 31	-2^{31} to $2^{31} - 1$
64	0 to 63	-2^{63} to $2^{63} - 1$

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

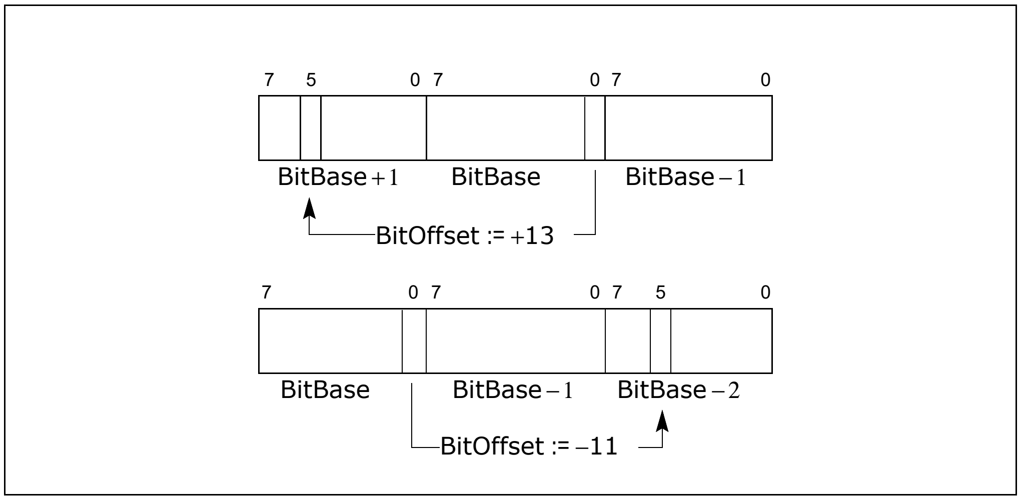


Figure 3-2. Memory Bit Indexing

3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section

The Intel C/C++ compiler intrinsic functions give access to the full power of the Intel Architecture Instruction Set, while allowing the compiler to optimize register allocation and instruction scheduling for faster execution. Most of these functions are associated with a single IA instruction, although some may generate multiple instructions or different instructions depending upon how they are used. In particular, these functions are used to invoke instructions that perform operations on vector registers that can hold multiple data elements. These SIMD instructions use the following data types.

- __m128, __m256, and __m512 can represent 4, 8, or 16 packed single precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128 data type is also used with various single precision floating-point scalar instructions that perform calculations using only the lowest 32 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.

- __m128d, __m256d, and __m512d can represent 2, 4, or 8 packed double precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128d data type is also used with various double precision floating-point scalar instructions that perform calculations using only the lowest 64 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- __m128i, __m256i, and __m512i can represent integer data in bytes, words, doublewords, quadwords, and occasionally larger data types.

Each of these data types incorporates in its name the number of bits it can hold. For example, the __m128 type holds 128 bits, and because each single precision floating-point value is 32 bits long the __m128 type holds (128/32) or four values. Normally the compiler will allocate memory for these data types on an even multiple of the size of the type. Such aligned memory locations may be faster to read and write than locations at other addresses.

These SIMD data types are not basic Standard C data types or C++ objects, so they may be used only with the assignment operator, passed as function arguments, and returned from a function call. If you access the internal members of these types directly, or indirectly by using them in a union, there may be side effects affecting optimization, so it is recommended to use them only with the SIMD instruction intrinsic functions described in this manual or the Intel C/C++ compiler documentation.

Many intrinsic functions names are prefixed with an indicator of the vector length and suffixed by an indicator of the vector element data type, although some functions do not follow the rules below. The prefixes are:

- _mm_ indicates that the function operates on 128-bit (or sometimes 64-bit) vectors.
- _mm256_ indicates the function operates on 256-bit vectors.
- _mm512_ indicates that the function operates on 512-bit vectors.

The suffixes include:

- _ps, which indicates a function that operates on packed single precision floating-point data. Packed single precision floating-point data corresponds to arrays of the C/C++ type *float* with either 4, 8 or 16 elements. Values of this type can be loaded from an array using the *_mm_loadu_ps, _mm256_loadu_ps,* or *_mm512_-loadu_ps* functions, or created from individual values using *_mm_set_ps, _mm256_set_ps,* or *_mm512_set_ps* functions, and they can be stored in an array using *_mm_storeu_ps, _mm256_storeu_ps,* or *_mm512_storeu_ps*.
- _ss, which indicates a function that operates on scalar single precision floating-point data. Single precision floating-point data corresponds to the C/C++ type *float*, and values of type float can be converted to type __m128 for use with these functions using the *_mm_set_ss* function, and converted back using the *_mm_cvtss_f32* function. When used with functions that operate on packed single precision floating-point data the scalar element corresponds with the first packed value.
- _pd, which indicates a function that operates on packed double precision floating-point data. Packed double precision floating-point data corresponds to arrays of the C/C++ type *double* with either 2, 4, or 8 elements. Values of this type can be loaded from an array using the *_mm_loadu_pd, _mm256_loadu_pd,* or *_mm512_-loadu_pd* functions, or created from individual values using *_mm_set_pd, _mm2566_set_pd,* or *_mm512_set_pd* functions, and they can be stored in an array using *_mm_storeu_pd, _mm256_storeu_pd,* or *_mm512_storeu_pd*.
- _sd, which indicates a function that operates on scalar double precision floating-point data. Double precision floating-point data corresponds to the C/C++ type *double*, and values of type double can be converted to type __m128d for use with these functions using the *_mm_set_sd* function, and converted back using the *_mm_cvtsd_f64* function. When used with functions that operate on packed double precision floating-point data the scalar element corresponds with the first packed value.
- _epi8, which indicates a function that operates on packed 8-bit signed integer values. Packed 8-bit signed integers correspond to an array of *signed char* with 16, 32 or 64 elements. Values of this type can be created from individual elements using *_mm_set_epi8, _mm256_set_epi8,* or *_mm512_set_epi8* functions.
- _epi16, which indicates a function that operates on packed 16-bit signed integer values. Packed 16-bit signed integers correspond to an array of *short* with 8, 16 or 32 elements. Values of this type can be created from individual elements using *_mm_set_epi16, _mm256_set_epi16,* or *_mm512_set_epi16* functions.

- _epi32, which indicates a function that operates on packed 32-bit signed integer values. Packed 32-bit signed integers correspond to an array of *int* with 4, 8 or 16 elements. Values of this type can be created from individual elements using *_mm_set_epi32, _mm256_set_epi32,* or *_mm512_set_epi32* functions.
- _epi64, which indicates a function that operates on packed 64-bit signed integer values. Packed 64-bit signed integers correspond to an array of *long long* (or *long* if it is a 64-bit data type) with 2, 4 or 8 elements. Values of this type can be created from individual elements using *_mm_set_epi32, _mm256_set_epi32,* or *_mm512_set_epi32* functions.
- _epu8, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of *unsigned char* with 16, 32 or 64 elements.
- _epu16, which indicates a function that operates on packed 16-bit unsigned integer values. Packed 16-bit unsigned integers correspond to an array of *unsigned short* with 8, 16 or 32 elements.
- _epu32, which indicates a function that operates on packed 32-bit unsigned integer values. Packed 32-bit unsigned integers correspond to an array of *unsigned* with 4, 8 or 16 elements.
- _epu64, which indicates a function that operates on packed 64-bit unsigned integer values. Packed 64-bit unsigned integers correspond to an array of *unsigned long long* (or *unsigned long* if it is a 64-bit data type) with 2, 4 or 8 elements.
- _si128, which indicates a function that operates on a single 128-bit value of type __m128i.
- _si256, which indicates a function that operates on a single a 256-bit value of type __m256i.
- _si512, which indicates a function that operates on a single a 512-bit value of type __m512i.

Values of any packed integer type can be loaded from an array using the *_mm_loadu_si128, _mm256_loadu_-si256,* or *_mm512_loadu_si512* functions, and they can be stored in an array using *_mm_storeu_si128, _mm256_storeu_si256,* or *_mm512_storeu_si512*.

These functions and data types are used with the SSE, AVX, and AVX-512 instruction set extension families. In addition there are similar functions that correspond to MMX instructions. These are less frequently used because they require additional state management, and only operate on 64-bit packed integer values.

The declarations of Intel C/C++ compiler intrinsic functions may reference some non-standard data types, such as __int64. The C Standard header *stdint.h* defines similar platform-independent types, and the documentation for that header gives characteristics that apply to corresponding non-standard types according to the following table.

Table 3-3. Standard and Non-Standard Data Types

Non-standard Type	Standard Type (from stdint.h)
__int64	int64_t
unsigned __int64	uint64_t
__int32	int32_t
unsigned __int32	uint32_t
__int16	int16_t
unsigned __int16	uint16_t

For a more detailed description of each intrinsic function and additional information related to its usage, refer to the online Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide.

3.1.1.11 Flags Affected Section

The “Flags Affected” section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, “EFLAGS Cross-Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Non-conventional assignments are described in the “Operation” section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

3.1.1.12 FPU Flags Affected Section

The floating-point instructions have an “FPU Flags Affected” section that describes how each instruction can affect the four condition code flags of the FPU status word.

3.1.1.13 Protected Mode Exceptions Section

The “Protected Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-4 associates each two-letter mnemonic with the corresponding exception vector and name. See Chapter 6, “Procedure Calls, Interrupts, and Exceptions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

Table 3-4. Intel 64® and IA-32 General Exceptions

Vector	Name	Source	Protected Mode[1]	Real Address Mode	Virtual 8086 Mode
0	#DE—Divide Error	DIV and IDIV instructions.	Yes	Yes	Yes
1	#DB—Debug	Any code or data reference.	Yes	Yes	Yes
3	#BP—Breakpoint	INT3 instruction.	Yes	Yes	Yes
4	#OF—Overflow	INTO instruction.	Yes	Yes	Yes
5	#BR—BOUND Range Exceeded	BOUND instruction.	Yes	Yes	Yes
6	#UD—Invalid Opcode (Undefined Opcode)	UD instruction or reserved opcode.	Yes	Yes	Yes
7	#NM—Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
8	#DF—Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Yes	Yes	Yes
10	#TS—Invalid TSS	Task switch or TSS access.	Yes	No	Yes
11	#NP—Segment Not Present	Loading segment registers or accessing system segments.	Yes	No	Yes
12	#SS—Stack Segment Fault	Stack operations and SS register loads.	Yes	Yes	Yes
13	#GP—General Protection[2]	Any memory reference and other protection checks.	Yes	Yes	Yes
14	#PF—Page Fault	Any memory reference.	Yes	No	Yes
16	#MF—Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
17	#AC—Alignment Check	Any data reference in memory.	Yes	No	Yes
18	#MC—Machine Check	Model dependent machine check errors.	Yes	Yes	Yes
19	#XM—SIMD Floating-Point Numeric Error	SSE/SSE2/SSE3 floating-point instructions.	Yes	Yes	Yes
20	#VE—Virtualization Exception	EPT violations[3]	Yes	No	No

Table 3-4. Intel 64® and IA-32 General Exceptions (Contd.)

Vector	Name	Source	Protected Mode[1]	Real Address Mode	Virtual 8086 Mode
21	#CP—Control Protection Exception	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.	Yes	No	No

NOTES:

1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.
3. This exception can occur only on processors that support the 1-setting of the "EPT-violation #VE" VM-execution control.

3.1.1.14 Real-Address Mode Exceptions Section

The "Real-Address Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-4).

3.1.1.15 Virtual-8086 Mode Exceptions Section

The "Virtual-8086 Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-4).

3.1.1.16 Floating-Point Exceptions Section

The "Floating-Point Exceptions" section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, exception 16) being generated. Table 3-5 associates a one- or two-letter mnemonic with the corresponding exception name. See "Floating-Point Exception Conditions" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a detailed description of these exceptions.

Table 3-5. x87 FPU Floating-Point Exceptions

Mnemonic	Name	Source
#IS #IA	Floating-point invalid operation: - Stack overflow or underflow - Invalid arithmetic operation	 - x87 FPU stack overflow or underflow - Invalid FPU arithmetic operation
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result (precision)	Inexact result (precision)

3.1.1.17 SIMD Floating-Point Exceptions Section

The "SIMD Floating-Point Exceptions" section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, exception 19) being generated. Table 3-6 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to "SSE and SSE2 Exceptions", in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Table 3-6. SIMD Floating-Point Exceptions

Mnemonic	Name	Source
#I	Floating-point invalid operation	Invalid arithmetic operation or source operand
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result	Inexact result (precision)

3.1.1.18 Compatibility Mode Exceptions Section

This section lists exceptions that occur within compatibility mode.

3.1.1.19 64-Bit Mode Exceptions Section

This section lists exceptions that occur within 64-bit mode.

3.2 INTEL® AMX CONSIDERATIONS

The following implementation parameters and helper functions are applicable to the Intel® AMX instructions.

3.2.1 Implementation Parameters

The parameters are reported via CPUID leaf 1DH. Index 0 reports all zeros for all fields.

```
define palette_table[id]:
    uint16_t total_tile_bytes
    uint16_t bytes_per_tile
    uint16_t bytes_per_row
    uint16_t max_names
    uint16_t max_rows
```

The tile parameters are set by LDTILECFG or XRSTOR* of TILECFG:

```
define tile[tid]:
    byte rows
    word colsb // bytes_per_row
    bool valid
```

3.2.2 Helper Functions

The helper functions used in Intel AMX instructions are defined below.

```
define write_row_and_zero(treg, r, data, nbytes):
   for j in 0 ...nbytes-1:
       treg.row[r].byte[j] := data.byte[j]

   // zero the rest of the row
   for j in nbytes ... palette_table[tilecfg.palette_id].bytes_per_row-1:
      treg.row[r].byte[j] := 0


define zero_upper_rows(treg, r):
   for i in r ... palette_table[tilecfg.palette_id].max_rows-1:
      for j in 0 ... palette_table[tilecfg.palette_id].bytes_per_row-1:
                   treg.row[i].byte[j] := 0



define zero_tilecfg_start():
   tilecfg.start_row :=0


define zero_all_tile_data():
  if XCR0[TILEDATA]:
     b :=CPUID(0xD,TILEDATA).EAX // size of feature
     for j in 0 ...b:
            TILEDATA.byte[j] := 0

define xcr0_supports_palette(palette_id):
  if palette_id == 0:
     return 1
  elif palette_id == 1:
     if XCR0[TILECFG] and XCR0[TILEDATA]:
        return 1
  return 0
```

3.3 INSTRUCTIONS (A-L)

The remainder of this chapter provides descriptions of Intel 64 and IA-32 instructions (A-L). See also: Chapter 4, “Instruction Set Reference, M-U,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B; Chapter 5, “Instruction Set Reference, V,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C; and Chapter 6, “Instruction Set Reference, W-Z,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

ADC—Add With Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 ib	ADC AL, imm8	I	Valid	Valid	Add with carry imm8 to AL.
15 iw	ADC AX, imm16	I	Valid	Valid	Add with carry imm16 to AX.
15 id	ADC EAX, imm32	I	Valid	Valid	Add with carry imm32 to EAX.
REX.W + 15 id	ADC RAX, imm32	I	Valid	N.E.	Add with carry imm32 sign extended to 64-bits to RAX.
80 /2 ib	ADC r/m8[1], imm8	MI	Valid	Valid	Add with carry imm8 to r/m8.
81 /2 iw	ADC r/m16, imm16	MI	Valid	Valid	Add with carry imm16 to r/m16.
81 /2 id	ADC r/m32, imm32	MI	Valid	Valid	Add with CF imm32 to r/m32.
REX.W + 81 /2 id	ADC r/m64, imm32	MI	Valid	N.E.	Add with CF imm32 sign extended to 64-bits to r/m64.
83 /2 ib	ADC r/m16, imm8	MI	Valid	Valid	Add with CF sign-extended imm8 to r/m16.
83 /2 ib	ADC r/m32, imm8	MI	Valid	Valid	Add with CF sign-extended imm8 into r/m32.
REX.W + 83 /2 ib	ADC r/m64, imm8	MI	Valid	N.E.	Add with CF sign-extended imm8 into r/m64.
10 /r	ADC r/m8[1], r8[1]	MR	Valid	Valid	Add with carry byte register to r/m8.
11 /r	ADC r/m16, r16	MR	Valid	Valid	Add with carry r16 to r/m16.
11 /r	ADC r/m32, r32	MR	Valid	Valid	Add with CF r32 to r/m32.
REX.W + 11 /r	ADC r/m64, r64	MR	Valid	N.E.	Add with CF r64 to r/m64.
12 /r	ADC r8[1], r/m8[1]	RM	Valid	Valid	Add with carry r/m8 to byte register.
13 /r	ADC r16, r/m16	RM	Valid	Valid	Add with carry r/m16 to r16.
13 /r	ADC r32, r/m32	RM	Valid	Valid	Add with CF r/m32 to r32.
REX.W + 13 /r	ADC r64, r/m64	RM	Valid	N.E.	Add with CF r/m64 to r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST + SRC + CF;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
ADC extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);
ADC extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);
ADC extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *sum_out);
ADC extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64
*sum_out);
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.
REX.W + 05 id	ADD RAX, imm32	I	Valid	N.E.	Add imm32 sign-extended to 64-bits to RAX.
80 /0 ib	ADD r/m8[1], imm8	MI	Valid	Valid	Add imm8 to r/m8.
81 /0 iw	ADD r/m16, imm16	MI	Valid	Valid	Add imm16 to r/m16.
81 /0 id	ADD r/m32, imm32	MI	Valid	Valid	Add imm32 to r/m32.
REX.W + 81 /0 id	ADD r/m64, imm32	MI	Valid	N.E.	Add imm32 sign-extended to 64-bits to r/m64.
83 /0 ib	ADD r/m16, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m16.
83 /0 ib	ADD r/m32, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m32.
REX.W + 83 /0 ib	ADD r/m64, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m64.
00 /r	ADD r/m8[1], r8[1]	MR	Valid	Valid	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8[1], r/m8[1]	RM	Valid	Valid	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST + SRC;
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADDPD—Add Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed double precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed double precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 58 /r VADDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.66.0F.W1 58 /r VADDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst {er}	C	V/V	AVX512F OR AVX10.1	Add packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds two, four or eight packed double precision floating-point values from the first source operand to the second source operand, and stores the packed double precision floating-point result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Vector Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]
      ELSE
         IF *merging-masking*              ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                           ; zeroing-masking
               DEST[i+63:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1)
            THEN
               DEST[i+63:i] := SRC1[i+63:i] + SRC2[63:0]
            ELSE
               DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]
         FI;
      ELSE
         IF *merging-masking*              ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                           ; zeroing-masking
               DEST[i+63:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VADDPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64] + SRC2[127:64]
DEST[191:128] := SRC1[191:128] + SRC2[191:128]
DEST[255:192] := SRC1[255:192] + SRC2[255:192]
DEST[MAXVL-1:256] := 0
.
```

VADDPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64] + SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ADDPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[127:64] := DEST[127:64] + SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDPD __m512d _mm512_add_pd (__m512d a, __m512d b);
VADDPD __m512d _mm512_mask_add_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VADDPD __m512d _mm512_maskz_add_pd (__mmask8 k, __m512d a, __m512d b);
VADDPD __m256d _mm256_mask_add_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VADDPD __m256d _mm256_maskz_add_pd (__mmask8 k, __m256d a, __m256d b);
VADDPD __m128d _mm_mask_add_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDPD __m128d _mm_maskz_add_pd (__mmask8 k, __m128d a, __m128d b);
VADDPD __m512d _mm512_add_round_pd (__m512d a, __m512d b, int);
VADDPD __m512d _mm512_mask_add_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VADDPD __m512d _mm512_maskz_add_round_pd (__mmask8 k, __m512d a, __m512d b, int);
ADDPD __m256d _mm256_add_pd (__m256d a, __m256d b);
ADDPD __m128d _mm_add_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

ADDPS—Add Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 58 /r ADDPS xmm1, xmm2/m128	A	V/V	SSE	Add packed single precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed single precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed single precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.128.0F.W0 58 /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.0F.W0 58 /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F OR AVX10.1	Add packed single precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds four, eight or sixteen packed single precision floating-point values from the first source operand with the second source operand, and stores the packed single precision floating-point result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] := SRC1[i+31:i] + SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VADDPS (VEX.256 Encoded Version)
```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[159:128] := SRC1[159:128] + SRC2[159:128]
DEST[191:160]:= SRC1[191:160] + SRC2[191:160]
DEST[223:192] := SRC1[223:192] + SRC2[223:192]
DEST[255:224] := SRC1[255:224] + SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VADDPS (VEX.128 Encoded Version)
```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ADDPS (128-bit Legacy SSE Version)
```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[63:32] := SRC1[63:32] + SRC2[63:32]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[127:96] := SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDPS __m512 _mm512_add_ps (__m512 a, __m512 b);
VADDPS __m512 _mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VADDPS __m512 _mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);
VADDPS __m256 _mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VADDPS __m256 _mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);
VADDPS __m128 _mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);
VADDPS __m128 _mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);
VADDPS __m512 _mm512_add_round_ps (__m512 a, __m512 b, int);
VADDPS __m512 _mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VADDPS __m512 _mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);
ADDPS __m256 _mm256_add_ps (__m256 a, __m256 b);
ADDPS __m128 _mm_add_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

ADDSD—Add Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LLIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Add the low double precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds the low double precision floating-point values from the second source operand and the first source operand and stores the double precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] + SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VADDSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

ADDSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDSD __m128d _mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d _mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d _mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

ADDSS—Add Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	A	V/V	SSE	Add the low single precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Add the low single precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LLIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Add the low single precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds the low single precision floating-point values from the second source operand and the first source operand, and stores the double precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:32) of the corresponding the destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSS (EVEX Encoded Versions)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] + SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VADDSS DEST, SRC1, SRC2 (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

ADDSS DEST, SRC (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] + SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

AESDEC—Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DE /r VAESDEC zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDECLAST instruction.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDEC

```
STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
STATE := InvMixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESDEC (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESDEC (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] :=0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)
VAESDEC __m256i _mm256_aesdec_epi128(__m256i, __m256i);
VAESDEC __m512i _mm512_aesdec_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-52, “Type E4NF Class Exception Conditions.”

AESDECLAST—Perform Last Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DF /r VAESDECLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using four128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

```
AESDECLAST
STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)

VAESDECLAST (128b and 256b VEX Encoded Versions)
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0

VAESDECLAST (EVEX Encoded Version)
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)
VAESDECLAST __m256i _mm256_aesdeclast_epi128(__m256i, __m256i);
VAESDECLAST __m512i _mm512_aesdeclast_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-52, “Type E4NF Class Exception Conditions.”

AESENC—Perform One Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DC /r AESENC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DC /r VAESENC zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform one round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from the zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a single round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENC-CLAST instruction.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENC

```
STATE := SRC1;
RoundKey := SRC2;
STATE := ShiftRows( STATE );
STATE := SubBytes( STATE );
STATE := MixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESENC (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR I := 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    STATE := MixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESENC (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i := 0 to KL-1:
    STATE := SRC1.xmm[i] // xmm[i] is the i'th xmm word in the SIMD register
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    STATE := MixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESENC __m128i _mm_aesenc (__m128i, __m128i)
VAESENC __m256i _mm256_aesenc_epi128(__m256i, __m256i);
VAESENC __m512i _mm512_aesenc_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

AESENCLAST—Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES (AVX512VL OR AVX10.1)	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DD /r VAESENCLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES (AVX512F OR AVX10.1)	Perform the last round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the last round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENCLAST

```
STATE := SRC1;
RoundKey := SRC2;
STATE := ShiftRows( STATE );
STATE := SubBytes( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

VAESENCLAST (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (1,128), (2,256)
FOR I=0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESENCLAST (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESENCLAST __m128i _mm_aesenclast (__m128i, __m128i)
VAESENCLAST __m256i _mm256_aesenclast_epi128(__m256i, __m256i);
VAESENCLAST __m512i _mm512_aesenclast_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-52, “Type E4NF Class Exception Conditions.”

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 ib	AND AL, imm8	I	Valid	Valid	AL AND imm8.
25 iw	AND AX, imm16	I	Valid	Valid	AX AND imm16.
25 id	AND EAX, imm32	I	Valid	Valid	EAX AND imm32.
REX.W + 25 id	AND RAX, imm32	I	Valid	N.E.	RAX AND imm32 sign-extended to 64-bits.
80 /4 ib	AND r/m8[1], imm8[1]	MI	Valid	Valid	r/m8 AND imm8.
81 /4 iw	AND r/m16, imm16	MI	Valid	Valid	r/m16 AND imm16.
81 /4 id	AND r/m32, imm32	MI	Valid	Valid	r/m32 AND imm32.
REX.W + 81 /4 id	AND r/m64, imm32	MI	Valid	N.E.	r/m64 AND imm32 sign extended to 64-bits.
83 /4 ib	AND r/m16, imm8	MI	Valid	Valid	r/m16 AND imm8 (sign-extended).
83 /4 ib	AND r/m32, imm8	MI	Valid	Valid	r/m32 AND imm8 (sign-extended).
REX.W + 83 /4 ib	AND r/m64, imm8	MI	Valid	N.E.	r/m64 AND imm8 (sign-extended).
20 /r	AND r/m8[1], r8[1]	MR	Valid	Valid	r/m8 AND r8.
21 /r	AND r/m16, r16	MR	Valid	Valid	r/m16 AND r16.
21 /r	AND r/m32, r32	MR	Valid	Valid	r/m32 AND r32.
REX.W + 21 /r	AND r/m64, r64	MR	Valid	N.E.	r/m64 AND r64.
22 /r	AND r8[1], r/m8[1]	RM	Valid	Valid	r8 AND r/m8.
23 /r	AND r16, r/m16	RM	Valid	Valid	r16 AND r/m16.
23 /r	AND r32, r/m32	RM	Valid	Valid	r32 AND r/m32.
REX.W + 23 /r	AND r64, r/m64	RM	Valid	N.E.	r64 AND r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the it to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST AND SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 55 /r VANDNPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 55 /r VANDNPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 55 /r VANDNPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND NOT of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND NOT of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VANDNPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VANDNPD (VEX.256 Encoded Version)

```
DEST[63:0] := (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] := (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[191:128] := (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
DEST[255:192] := (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VANDNPD (VEX.128 Encoded Version)

```
DEST[63:0] := (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] := (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ANDNPD (128-bit Legacy SSE Version)

```
DEST[63:0] := (NOT(DEST[63:0])) BITWISE AND SRC[63:0]
DEST[127:64] := (NOT(DEST[127:64])) BITWISE AND SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VANDNPD __m512d _mm512_andnot_pd (__m512d a, __m512d b);
VANDNPD __m512d _mm512_mask_andnot_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDNPD __m512d _mm512_maskz_andnot_pd (__mmask8 k, __m512d a, __m512d b);
VANDNPD __m256d _mm256_mask_andnot_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDNPD __m256d _mm256_maskz_andnot_pd (__mmask8 k, __m256d a, __m256d b);
VANDNPD __m128d _mm_mask_andnot_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDNPD __m128d _mm_maskz_andnot_pd (__mmask8 k, __m128d a, __m128d b);
VANDNPD __m256d _mm256_andnot_pd (__m256d a, __m256d b);
ANDNPD __m128d _mm_andnot_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 55 /r ANDNPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 55 /r VANDNPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 55 /r VANDNPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND NOT of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDNPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
               THEN
                  DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]
               ELSE
                  DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]
            FI;
         ELSE
            IF *merging-masking*                 ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE                              ; zeroing-masking
                  DEST[i+31:i] = 0
            FI;
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VANDNPS (VEX.256 Encoded Version)

```
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] := (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[159:128] := (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]
DEST[191:160] := (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]
DEST[223:192] := (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]
DEST[255:224] := (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VANDNPS (VEX.128 Encoded Version)

```
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] := (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ANDNPS (128-bit Legacy SSE Version)

```
DEST[31:0] := (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] := (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] := (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] := (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS __m512 _mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 _mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 _mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 _mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 _mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 _mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 _mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 b);
ANDNPS __m128 _mm_andnot_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VANDPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VANDPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VANDPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ANDPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE AND SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE AND SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VANDPD __m512d _mm512_and_pd (__m512d a, __m512d b);
VANDPD __m512d _mm512_mask_and_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDPD __m512d _mm512_maskz_and_pd (__mmask8 k, __m512d a, __m512d b);
VANDPD __m256d _mm256_mask_and_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDPD __m256d _mm256_maskz_and_pd (__mmask8 k, __m256d a, __m256d b);
VANDPD __m128d _mm_mask_and_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDPD __m128d _mm_maskz_and_pd (__mmask8 k, __m128d a, __m128d b);
VANDPD __m256d _mm256_and_pd (__m256d a, __m256d b);
ANDPD __m128d _mm_and_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 54 /r ANDPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 54 /r VANDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 54 /r VANDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+31:i] BITWISE AND SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

VANDPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE AND SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE AND SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE AND SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VANDPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] := 0;
```

ANDPS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] BITWISE AND SRC[31:0]
DEST[63:32] := DEST[63:32] BITWISE AND SRC[63:32]
DEST[95:64] := DEST[95:64] BITWISE AND SRC[95:64]
DEST[127:96] := DEST[127:96] BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDPS __m512 _mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 _mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 _mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 _mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 _mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 _mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 _mm_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 _mm256_and_ps (__m256 a, __m256 b);
ANDPS __m128 _mm_and_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

BSF—Bit Scan Forward

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BC /r	BSF r16, r/m16	RM	Valid	Valid	Bit scan forward on r/m16.
0F BC /r	BSF r32, r/m32	RM	Valid	Valid	Bit scan forward on r/m32.
REX.W + 0F BC /r	BSF r64, r/m64	RM	Valid	N.E.	Bit scan forward on r/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is zero, the destination operand is unmodified.[1]

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF SRC <> 0
    THEN
        temp := 0;
        WHILE Bit(SRC, temp) = 0
        DO
            temp := temp + 1;
        OD;
        DEST := temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The PF flag is set to 1 if the number of bits set in the source operand is even; otherwise, it is cleared. The CF, OF, SF, and AF flags are all cleared.[2]

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

1. With a zero source operand on some older processors, use of a 32-bit operand size may clear the upper 32 bits of a 64-bit destination while leaving the lower 32 bits unmodified.
2. On some older processors, the CF, OF, SF, AF, and PF flags are unmodified.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

BSR—Bit Scan Reverse

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BD /r	BSR r16, r/m16	RM	Valid	Valid	Bit scan reverse on r/m16.
0F BD /r	BSR r32, r/m32	RM	Valid	Valid	Bit scan reverse on r/m32.
REX.W + 0F BD /r	BSR r64, r/m64	RM	Valid	N.E.	Bit scan reverse on r/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is zero, the destination operand is unmodified.[1]

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF SRC <> 0
        temp := OperandSize – 1;
        WHILE Bit(SRC, temp) = 0
        DO
            temp := temp - 1;
        OD;
        DEST := temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The PF flag is set to 1 if the number of bits set in the source operand is even; otherwise, it is cleared. The CF, OF, SF, and AF flags are all cleared.[2]

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

1. With a zero source operand on some older processors, use of a 32-bit operand size may clear the upper 32 bits of a 64-bit destination while leaving the lower 32 bits unmodified.
2. On some older processors, the CF, OF, SF, AF, and PF flags are unmodified.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

BTC—Bit Test and Complement

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BB /r	BTC r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and complement.
0F BB /r	BTC r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + 0F BB /r	BTC r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and complement.
0F BA /7 ib	BTC r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and complement.
0F BA /7 ib	BTC r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + 0F BA /7 ib	BTC r/m64, imm8	MI	Valid	N.E.	Store selected bit in CF flag and complement.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15) for the bit base. Using a REX prefix in the form of REX.R permits access to R8-R15 for the bit offset (when it uses a register). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
CF := Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) := NOT Bit(BitBase, BitOffset);
```

Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

CMP—Compare Two Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
3C ib	CMP AL, imm8	I	Valid	Valid	Compare imm8 with AL.
3D iw	CMP AX, imm16	I	Valid	Valid	Compare imm16 with AX.
3D id	CMP EAX, imm32	I	Valid	Valid	Compare imm32 with EAX.
REX.W + 3D id	CMP RAX, imm32	I	Valid	N.E.	Compare imm32 sign-extended to 64-bits with RAX.
80 /7 ib	CMP r/m8[1], imm8	MI	Valid	Valid	Compare imm8 with r/m8.
81 /7 iw	CMP r/m16, imm16	MI	Valid	Valid	Compare imm16 with r/m16.
81 /7 id	CMP r/m32, imm32	MI	Valid	Valid	Compare imm32 with r/m32.
REX.W + 81 /7 id	CMP r/m64, imm32	MI	Valid	N.E.	Compare imm32 sign-extended to 64-bits with r/m64.
83 /7 ib	CMP r/m16, imm8	MI	Valid	Valid	Compare imm8 with r/m16.
83 /7 ib	CMP r/m32, imm8	MI	Valid	Valid	Compare imm8 with r/m32.
REX.W + 83 /7 ib	CMP r/m64, imm8	MI	Valid	N.E.	Compare imm8 with r/m64.
38 /r	CMP r/m8[1], r8[1]	MR	Valid	Valid	Compare r8 with r/m8.
39 /r	CMP r/m16, r16	MR	Valid	Valid	Compare r16 with r/m16.
39 /r	CMP r/m32, r32	MR	Valid	Valid	Compare r32 with r/m32.
REX.W + 39 /r	CMP r/m64,r64	MR	Valid	N.E.	Compare r64 with r/m64.
3A /r	CMP r8[1], r/m8[1]	RM	Valid	Valid	Compare r/m8 with r8.
3B /r	CMP r16, r/m16	RM	Valid	Valid	Compare r/m16 with r16.
3B /r	CMP r32, r/m32	RM	Valid	Valid	Compare r/m32 with r32.
REX.W + 3B /r	CMP r64, r/m64	RM	Valid	N.E.	Compare r/m64 with r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX (r)	imm8/16/32	N/A	N/A

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, “EFLAGS Condition Codes,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, shows the relationship of the status flags and the condition codes.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
temp := SRC1 − SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

CMPPD—Compare Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Compare packed double precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.66.0F.WIG C2 /r ib VCMPPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed double precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.66.0F.WIG C2 /r ib VCMPPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed double precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed double precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed double precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, zmm2, zmm3/m512/m64bcst {sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare packed double precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

Table 3-8. Comparison Predicate for CMPPD and CMPPS Instructions

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes

Table 3-8. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
NGE_UQ	19H	Not-greater-than-or-equal (unordered, non-signaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignal-ing)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

NOTES:

1. If either operand A or B is a NAN.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-9. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-9. Pseudo-Op and CMPPD Implementation

Pseudo-Op	CMPPD Implementation
CMPEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 0
CMPLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 1
CMPLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 2
CMPUNORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 3
CMPNEQPD xmm1, xmm2	CMPPD xmm1, xmm2, 4
CMPNLTPD xmm1, xmm2	CMPPD xmm1, xmm2, 5
CMPNLEPD xmm1, xmm2	CMPPD xmm1, xmm2, 6
CMPORDPD xmm1, xmm2	CMPPD xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-10, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-10, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 3-10.

Table 3-10. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	**CMPPD Implementation**
VCMPEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0*
VCMPLTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1*
VCMPLEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 2*
VCMPUNORDPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 3*
VCMPNEQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 4*
VCMPNLTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 5*
VCMPNLEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 6*
VCMPORDPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 7*
VCMPEQ_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 8*
VCMPNGEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 9*
VCMPNGTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0AH*
VCMPFALSEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0CH*
VCMPGEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0DH*
VCMPGTPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0EH*
VCMPTRUEPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 10H*
VCMPLT_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 11H*
VCMPLE_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 12H*
VCMPUNORD_SPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 13H*
VCMPNEQ_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 14H*
VCMPNLT_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 15H*
VCMPNLE_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 16H*
VCMPORD_SPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 17H*
VCMPEQ_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 18H*
VCMPNGE_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 19H*
VCMPNGT_UQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1CH*
VCMPGE_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1DH*
VCMPGT_OQPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPD *reg1, reg2, reg3*	VCMPPD *reg1, reg2, reg3, 1FH*

Operation

```
CASE (COMPARISON PREDICATE) OF
0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved;
ESAC;
```

VCMPPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    CMP := SRC1[i+63:i] OP5 SRC2[63:0]
                ELSE
                    CMP := SRC1[i+63:i] OP5 SRC2[i+63:i]
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0            ; zeroing-masking only
    FI;
```

```
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VCMPPD (VEX.256 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
CMP2 := SRC1[191:128] OP5 SRC2[191:128];
CMP3 := SRC1[255:192] OP5 SRC2[255:192];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[191:128] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[191:128] := 0000000000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[255:192] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[255:192] := 0000000000000000H; FI;
DEST[MAXVL-1:256] := 0
```

VCMPPD (VEX.128 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] := 0
```

CMPPD (128-bit Legacy SSE Version)

```
CMP0 := SRC1[63:0] OP3 SRC2[63:0];
CMP1 := SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPPD __mmask8 _mm512_cmp_pd_mask( __m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_cmp_round_pd_mask( __m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm512_mask_cmp_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_mask_cmp_round_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm256_cmp_pd_mask( __m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm256_mask_cmp_pd_mask( __mmask8 k1, __m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm_cmp_pd_mask( __m128d a, __m128d b, int imm);
VCMPPD __mmask8 _mm_mask_cmp_pd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPPD __m256 _mm256_cmp_pd(__m256d a, __m256d b, int imm)
```

(V)CMPPD __m128 _mm_cmp_pd(__m128d a, __m128d b, int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

CMPPS—Compare Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	A	V/V	SSE	Compare packed single precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed single precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed single precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.0F.W0 C2 /r ib VCMPPS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed single precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.0F.W0 C2 /r ib VCMPPS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed single precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.0F.W0 C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst {sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare packed single precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-11. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-11. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 0
CMPLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 1
CMPLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 2
CMPUNORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 3
CMPNEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 4
CMPNLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 5
CMPNLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 6
CMPORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-12, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-12, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-12.

Table 3-12. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0*
VCMPLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1*
VCMPLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 2*
VCMPUNORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 3*
VCMPNEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 4*
VCMPNLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 5*
VCMPNLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 6*
VCMPORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 7*
VCMPEQ_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 8*
VCMPNGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 9*
VCMPNGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0AH*
VCMPFALSEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0CH*
VCMPGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0DH*
VCMPGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0EH*
VCMPTRUEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 10H*
VCMPLT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 11H*
VCMPLE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 12H*
VCMPUNORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 13H*
VCMPNEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 14H*
VCMPNLT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 15H*
VCMPNLE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 16H*
VCMPORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 17H*
VCMPEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 18H*
VCMPNGE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 19H*
VCMPNGT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1CH*
VCMPGE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1DH*
VCMPGT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1FH*

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    CMP := SRC1[i+31:i] OP5 SRC2[31:0]
                ELSE
                    CMP := SRC1[i+31:i] OP5 SRC2[i+31:i]
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking onlyFI;
    FI;
```

```
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VCMPPS (VEX.256 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
CMP1 := SRC1[63:32] OP5 SRC2[63:32];
CMP2 := SRC1[95:64] OP5 SRC2[95:64];
CMP3 := SRC1[127:96] OP5 SRC2[127:96];
CMP4 := SRC1[159:128] OP5 SRC2[159:128];
CMP5 := SRC1[191:160] OP5 SRC2[191:160];
CMP6 := SRC1[223:192] OP5 SRC2[223:192];
CMP7 := SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] :=000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 000000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] := FFFFFFFFH;
    ELSE DEST[159:128] := 000000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] := FFFFFFFFH;
    ELSE DEST[191:160] := 000000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] := FFFFFFFFH;
    ELSE DEST[223:192] :=000000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] := FFFFFFFFH;
    ELSE DEST[255:224] := 000000000H; FI;
DEST[MAXVL-1:256] := 0
```

VCMPPS (VEX.128 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
CMP1 := SRC1[63:32] OP5 SRC2[63:32];
CMP2 := SRC1[95:64] OP5 SRC2[95:64];
CMP3 := SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] := 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] :=000000000H; FI;
DEST[MAXVL-1:128] := 0
```

CMPPS (128-bit Legacy SSE Version)

```
CMP0 := SRC1[31:0] OP3 SRC2[31:0];
CMP1 := SRC1[63:32] OP3 SRC2[63:32];
CMP2 := SRC1[95:64] OP3 SRC2[95:64];
CMP3 := SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] :=FFFFFFFFH;
    ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] := 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] := FFFFFFFFH;
    ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] :=000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPPS __mmask16 _mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 _mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask8 _mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm_cmp_ps_mask( __m128 a, __m128 b, int imm);
VCMPPS __mmask8 _mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 _mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPPS __m128 _mm_cmp_ps(__m128 a, __m128 b, int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

CMPSD—Compare Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	A	V/V	SSE2	Compare low double precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F2.0F.W1 C2 /r ib VCMPSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX =0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-13. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-13. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 0
CMPLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 1
CMPLESD xmm1, xmm2	CMPSD xmm1, xmm2, 2
CMPUNORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 3
CMPNEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 4
CMPNLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 5
CMPNLESD xmm1, xmm2	CMPSD xmm1, xmm2, 6
CMPORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-14, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-14, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-14.

Table 3-14. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0
VCMPLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1
VCMPLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 2
VCMPUNORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 3
VCMPNEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 4
VCMPNLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 5
VCMPNLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 6
VCMPORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 7
VCMPEQ_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 8
VCMPNGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 9
VCMPNGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0AH
VCMPFALSESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0BH
VCMPNEQ_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0CH
VCMPGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0DH

Table 3-14. Pseudo-Op and VCMPSD Implementation (Contd.)

Pseudo-Op	CMPSD Implementation
VCMPGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0EH
VCMPTRUESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0FH
VCMPEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 10H
VCMPLT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 11H
VCMPLE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 12H
VCMPUNORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 13H
VCMPNEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 14H
VCMPNLT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 15H
VCMPNLE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 16H
VCMPORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 17H
VCMPEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 18H
VCMPNGE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 19H
VCMPNGT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1AH
VCMPFALSE_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1BH
VCMPNEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1CH
VCMPGE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1DH
VCMPGT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1EH
VCMPTRUE_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1FH

Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSD (EVEX Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                    ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

CMPSD (128-bit Legacy SSE Version)

```
CMP0 := DEST[63:0] OP3 SRC[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[MAXVL-1:64] (Unmodified)
```

VCMPSD (VEX.128 Encoded Version)

```
CMP0 := SRC1[63:0] OP5 SRC2[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPSD __mmask8 _mm_cmp_sd_mask( __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_cmp_round_sd_mask( __m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 _mm_mask_cmp_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_mask_cmp_round_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d _mm_cmp_sd(__m128d a, __m128d b, const int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

CMPSS—Compare Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	A	V/V	SSE	Compare low single precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	C	V/V	AVX512F OR AVX10.1	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 127:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-8). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-8). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-15. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-15. Pseudo-Op and CMPSS Implementation

Pseudo-Op	CMPSS Implementation
CMPEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 0
CMPLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 1
CMPLESS xmm1, xmm2	CMPSS xmm1, xmm2, 2
CMPUNORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 3
CMPNEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 4
CMPNLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 5
CMPNLESS xmm1, xmm2	CMPSS xmm1, xmm2, 6
CMPORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 7

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-14, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 3-16, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSS instructions in a similar fashion by extending the syntax listed in Table 3-16.

Table 3-16. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPEQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0
VCMPLTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1
VCMPLESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 2
VCMPUNORDSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 3
VCMPNEQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 4
VCMPNLTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 5
VCMPNLESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 6
VCMPORDSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 7
VCMPEQ_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 8
VCMPNGESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 9
VCMPNGTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0AH
VCMPFALSESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0BH
VCMPNEQ_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0CH

Table 3-16. Pseudo-Op and VCMPSS Implementation (Contd.)

Pseudo-Op	CMPSS Implementation
VCMPGESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0DH
VCMPGTSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0EH
VCMPTRUESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0FH
VCMPEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 10H
VCMPLT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 11H
VCMPLE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 12H
VCMPUNORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 13H
VCMPNEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 14H
VCMPNLT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 15H
VCMPNLE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 16H
VCMPORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 17H
VCMPEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 18H
VCMPNGE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 19H
VCMPNGT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1AH
VCMPFALSE_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1BH
VCMPNEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1CH
VCMPGE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1DH
VCMPGT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1EH
VCMPTRUE_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1FH

Software should ensure VCMPSS is encoded with VEX.L=0. Encoding VCMPSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 := EQ_OQ; OP5 := EQ_OQ;
    1: OP3 := LT_OS; OP5 := LT_OS;
    2: OP3 := LE_OS; OP5 := LE_OS;
    3: OP3 := UNORD_Q; OP5 := UNORD_Q;
    4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
    5: OP3 := NLT_US; OP5 := NLT_US;
    6: OP3 := NLE_US; OP5 := NLE_US;
    7: OP3 := ORD_Q; OP5 := ORD_Q;
    8: OP5 := EQ_UQ;
    9: OP5 := NGE_US;
    10: OP5 := NGT_US;
    11: OP5 := FALSE_OQ;
    12: OP5 := NEQ_OQ;
    13: OP5 := GE_OS;
    14: OP5 := GT_OS;
    15: OP5 := TRUE_UQ;
    16: OP5 := EQ_OS;
    17: OP5 := LT_OQ;
    18: OP5 := LE_OQ;
    19: OP5 := UNORD_S;
    20: OP5 := NEQ_US;
    21: OP5 := NLT_UQ;
    22: OP5 := NLE_UQ;
    23: OP5 := ORD_S;
    24: OP5 := EQ_US;
    25: OP5 := NGE_UQ;
    26: OP5 := NGT_UQ;
    27: OP5 := FALSE_OS;
    28: OP5 := NEQ_OS;
    29: OP5 := GE_OQ;
    30: OP5 := GT_OQ;
    31: OP5 := TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSS (EVEX Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

CMPSS (128-bit Legacy SSE Version)

```
CMP0 := DEST[31:0] OP3 SRC[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; FI;
DEST[MAXVL-1:32] (Unmodified)
```

VCMPSS (VEX.128 Encoded Version)

```
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPSS __mmask8 _mm_cmp_ss_mask( __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_cmp_round_ss_mask( __m128 a, __m128 b, int imm, int sae);
VCMPSS __mmask8 _mm_mask_cmp_ss_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_mask_cmp_round_ss_mask( __mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-8, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

CMPXCHG—Compare and Exchange

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B0/r	CMPXCHG r/m8[1], r8[1]	MR	Valid	Valid[2]	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
0F B1/r	CMPXCHG r/m16, r16	MR	Valid	Valid[2]	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AX.
0F B1/r	CMPXCHG r/m32, r32	MR	Valid	Valid[2]	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into EAX.
REX.W + 0F B1/r	CMPXCHG r/m64, r64	MR	Valid	N.E.	Compare RAX with r/m64. If equal, ZF is set and r64 is loaded into r/m64. Else, clear ZF and load r/m64 into RAX.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A

Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

Operation

```
(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed *)
TEMP := DEST
IF accumulator = TEMP
    THEN
        ZF := 1;
        DEST := SRC;
    ELSE
        ZF := 0;
        accumulator := TEMP;
        DEST := TEMP;
FI;
```

Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

COMISD—Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LLIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F OR AVX10.1	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Compares the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid operation exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISD (All Versions)

```
RESULT :=    OrderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd (__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comile_sd (__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comige_sd (__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2F /r VCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LLIG.0F.W0 2F /r VCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F OR AVX10.1	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Compares the single precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid operation exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISS (All Versions)

```
RESULT :=   OrderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
VCOMISS int _mm_comieq_ss (__m128 a, __m128 b)
VCOMISS int _mm_comilt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comile_ss (__m128 a, __m128 b)
VCOMISS int _mm_comigt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comige_ss (__m128 a, __m128 b)
VCOMISS int _mm_comineq_ss (__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CPUID—CPU Identification

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	ZO	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-17 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using some Intel processors, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)[2]
CPUID.EAX =1FH (* Returns V2 Extended Topology Enumeration leaf. *)[2]
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

"Serializing Instructions" in Chapter 10, "Multiple-Processor Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

"Caching Translation Information" in Chapter 4, "Linear-Address Pre-Processing," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.
2. CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of CPUID leaf 1FH before using leaf 0BH.

Table 3-17. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	“Genu”
	ECX	“ntel”
	EDX	“ineI”
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).
	EBX	Bits 07-00: Brand Index. Bits 15-08: CLFLUSH line size (Value ∗ 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID**.
	ECX	Feature Information (see Figure 3-7 and Table 3-19).
	EDX	Feature Information (see Figure 3-8 and Table 3-20). **NOTES:** * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1. ** *The 8-bit initial APIC ID in EBX[31:24] is replaced by the 32-bit x2APIC ID, available in Leaf 0BH and Leaf 1FH.*
02H	EAX	Cache and TLB Information (see Table 3-21).
	EBX	Cache and TLB Information.
	ECX	Cache and TLB Information.
	EDX	Cache and TLB Information.
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00-31 of 96-bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32-63 of 96-bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) **NOTES:** Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
	Deterministic Cache Parameters Leaf (Initial EAX Value = 04H)	
04H		**NOTES:** Leaf 04H output depends on the initial value in ECX.* See also: “INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level” on page 256.
	EAX	Bits 04-00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		Bits 07-05: Cache Level (starts at 1). Bit 08: Self Initializing cache level (does not need SW initialization). Bit 09: Fully Associative cache. Bits 13-10: Reserved. Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache**, ***. Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****.
	EBX	Bits 11-00: L = System Coherency Line Size**. Bits 21-12: P = Physical Line partitions**. Bits 31-22: W = Ways of associativity**.
	ECX	Bits 31-00: S = Number of Sets**.
	EDX	Bit 00: Write-Back Invalidate/Invalidate. 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache. 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 02: Complex Cache Indexing. 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31-03: Reserved = 0. **NOTES:** * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0. ** Add one to the return value to get the result. ***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache. **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
	MONITOR/MWAIT Leaf (Initial EAX Value = 05H)	
05H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity). Bits 31-16: Reserved = 0.
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity). Bits 31-16: Reserved = 0.
	ECX	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported. Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled. Bits 31-02: Reserved.
	EDX	Bits 03-00: Number of C0* sub C-states supported using MWAIT. Bits 07-04: Number of C1* sub C-states supported using MWAIT. Bits 11-08: Number of C2* sub C-states supported using MWAIT. Bits 15-12: Number of C3* sub C-states supported using MWAIT. Bits 19-16: Number of C4* sub C-states supported using MWAIT. Bits 23-20: Number of C5* sub C-states supported using MWAIT. Bits 27-24: Number of C6* sub C-states supported using MWAIT. Bits 31-28: Number of C7* sub C-states supported using MWAIT.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		NOTE: * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
	Thermal and Power Management Leaf (Initial EAX Value = 06H)	
06H	EAX	Bit 00: Digital temperature sensor is supported if set. Bit 01: Intel Turbo Boost Technology available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved. Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bit 14: Intel® Turbo Boost Max Technology 3.0 available. Bit 15: HWP Capabilities. Highest Performance change is supported if set. Bit 16: HWP PECI override is supported if set. Bit 17: Flexible HWP is supported if set. Bit 18: Fast access mode, low latency, and posted IA32_HWP_REQUEST MSR are supported if set. Bit 19: HW_FEEDBACK. IA32_HW_FEEDBACK_PTR MSR, IA32_HW_FEEDBACK_CONFIG MSR, IA32_PACKAGE_THERM_STATUS MSR bit 26, and IA32_PACKAGE_THERM_INTERRUPT MSR bit 25 are supported if set. Bit 20: Ignoring Idle Logical Processor HWP request is supported if set. Bit 21: Reserved. Bit 22: HWP Control MSR Support. The IA32_HWP_CTL MSR is supported if set. Bit 23: Intel® Thread Director supported if set. The IA32_HW_FEEDBACK_CHAR and IA32_HW_FEEDBACK_THREAD_CONFIG MSRs are supported if set. Bit 24: IA32_THERM_INTERRUPT MSR bit 25 is supported if set. Bits 31-25: Reserved.
	EBX	Bits 03-00: Number of Interrupt Thresholds in Digital Thermal Sensor. Bits 31-04: Reserved.
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. Bits 02-01: Reserved = 0. Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H). Bits 07-04: Reserved = 0. Bits 15-08: Number of Intel® Thread Director classes supported by the processor. Information for that many classes is written into the Intel Thread Director Table by the hardware. Bits 31-16: Reserved = 0.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 07-00: Bitmap of supported hardware feedback interface capabilities. 0 = When set to 1, indicates support for performance capability reporting. 1 = When set to 1, indicates support for energy efficiency capability reporting. 2-7 = Reserved Bits 11-08: Enumerates the size of the hardware feedback interface structure in number of 4 KB pages; add one to the return value to get the result. Bits 31-16: Index (starting at 0) of this logical processor's row in the hardware feedback interface structure. Note that on some parts the index may be same for multiple logical processors. On some parts the indices may not be contiguous, i.e., there may be unused rows in the hardware feedback interface structure. **NOTE:** Bits 0 and 1 will always be set together.
	Structured Extended Feature Flags Enumeration Leaf (Initial EAX Value = 07H, ECX = 0)	
07H	EAX	Bits 31-00: Reports the maximum input value for supported leaf 7 sub-leaves.
	EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1. Bit 03: BMI1. Bit 04: HLE. Bit 05: AVX2. Supports Intel® Advanced Vector Extensions 2 (Intel® AVX2) if 1. Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1. Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1. Bit 08: BMI2. Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers. Bit 11: RTM. Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1. Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1. Bit 16: AVX512F. Bit 17: AVX512DQ. Bit 18: RDSEED. Bit 19: ADX. Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1. Bit 21: AVX512_IFMA. Bit 22: Reserved. Bit 23: CLFLUSHOPT. Bit 24: CLWB. Bit 25: Intel Processor Trace. Bit 26: AVX512PF. (Intel® Xeon Phi™ only.) Bit 27: AVX512ER. (Intel® Xeon Phi™ only.) Bit 28: AVX512CD. Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1. Bit 30: AVX512BW. Bit 31: AVX512VL.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bit 00: PREFETCHWT1. (Intel® Xeon Phi™ only.) Bit 01: AVX512_VBMI. Bit 02: UMIP. Supports user-mode instruction prevention if 1. Bit 03: PKU. Supports protection keys for user-mode pages if 1. Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions). Bit 05: WAITPKG. Bit 06: AVX512_VBMI2. Bit 07: CET_SS. Supports CET shadow stack features if 1. Processors that set this bit define bits 1:0 of the IA32_U_CET and IA32_S_CET MSRs. Enumerates support for the following MSRs: IA32_INTERRUPT_SSP_TABLE_ADDR, IA32_PL3_SSP, IA32_PL2_SSP, IA32_PL1_SSP, and IA32_PL0_SSP. Bit 08: GFNI. Bit 09: VAES. Bit 10: VPCLMULQDQ. Bit 11: AVX512_VNNI. Bit 12: AVX512_BITALG. Bits 13: TME_EN. If 1, the following MSRs are supported: IA32_TME_CAPABILITY, IA32_TME_ACTIVATE, IA32_TME_EXCLUDE_MASK, and IA32_TME_EXCLUDE_BASE. Bit 14: AVX512_VPOPCNTDQ. Bit 15: Reserved. Bit 16: LA57. Supports 57-bit linear addresses and five-level paging if 1. Bits 21-17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode. Bit 22: RDPID and IA32_TSC_AUX are available if 1. Bit 23: KL. Supports Key Locker if 1. Bit 24: BUS_LOCK_DETECT. If 1, indicates support for OS bus-lock detection. Bit 25: CLDEMOTE. Supports cache line demote if 1. Bit 26: Reserved. Bit 27: MOVDIRI. Supports MOVDIRI if 1. Bit 28: MOVDIR64B. Supports MOVDIR64B if 1. Bit 29: ENQCMD. Supports Enqueue Stores if 1. Bit 30: SGX_LC. Supports SGX Launch Configuration if 1. Bit 31: PKS. Supports protection keys for supervisor-mode pages if 1.
	EDX	Bit 00: Reserved. Bit 01: SGX-KEYS. If 1, Attestation Services for Intel® SGX is supported. Bit 02: AVX512_4VNNIW. (Intel® Xeon Phi™ only.) Bit 03: AVX512_4FMAPS. (Intel® Xeon Phi™ only.) Bit 04: Fast Short REP MOV. Bit 05: UINTR. If 1, the processor supports user interrupts. Bits 07-06: Reserved. Bit 08: AVX512_VP2INTERSECT. Bit 09: SRBDS_CTRL. If 1, enumerates support for the IA32_MCU_OPT_CTRL MSR and indicates its bit 0 (RNGDS_MITG_DIS) is also supported. Bit 10: MD_CLEAR supported. Bit 11: RTM_ALWAYS_ABORT. If set, any execution of XBEGIN immediately aborts and transitions to the specified fallback address. Bit 12: Reserved. Bit 13: If 1, RTM_FORCE_ABORT supported. Processors that set this bit support the IA32_TSX_FORCE_ABORT MSR. They allow software to set IA32_TSX_FORCE_ABORT[0] (RTM_FORCE_ABORT). Bit 14: SERIALIZE. Bit 15: Hybrid. If 1, the processor is identified as a hybrid part. If CPUID.0.MAXLEAF $\geq$ 1AH and CPUID.1A.EAX $\neq$ 0, then the Native Model ID Enumeration Leaf 1AH exists. Bit 16: TSXLDTRK. If 1, the processor supports Intel TSX suspend/resume of load address tracking.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Bit 17: Reserved. Bit 18: PCONFIG. Supports PCONFIG if 1. Bit 19: Architectural LBRs. If 1, indicates support for architectural LBRs. Bit 20: CET_IBT. Supports CET indirect branch tracking features if 1. Processors that set this bit define bits 5:2 and bits 63:10 of the IA32_U_CET and IA32_S_CET MSRs. Bit 21: Reserved. Bit 22: AMX-BF16. If 1, the processor supports tile computational operations on bfloat16 numbers. Bit 23: AVX512-FP16. Bit 24: AMX-TILE. If 1, the processor supports tile architecture. Bits 25: AMX-INT8. If 1, the processor supports tile computational operations on 8-bit integers. Bit 26: Enumerates support for indirect branch restricted speculation (IBRS) and the indirect branch predictor barrier (IBPB). Processors that set this bit support the IA32_SPEC_CTRL MSR and the IA32_PRED_CMD MSR. They allow software to set IA32_SPEC_CTRL[0] (IBRS) and IA32_PRED_CMD[0] (IBPB). Bit 27: Enumerates support for single thread indirect branch predictors (STIBP). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[1] (STIBP). Bit 28: Enumerates support for L1D_FLUSH. Processors that set this bit support the IA32_FLUSH_CMD MSR. They allow software to set IA32_FLUSH_CMD[0] (L1D_FLUSH). Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR. Bit 30: Enumerates support for the IA32_CORE_CAPABILITIES MSR. IA32_CORE_CAPABILITIES is an architectural MSR that enumerates model-specific features. A bit being set in this MSR indicates that a model specific feature is supported; software must still consult CPUID family/model/stepping to determine the behavior of the enumerated feature as features enumerated in IA32_CORE_CAPABILITIES may have different behavior on different processor models. Some of these features may have behavior that is consistent across processor models (and for which consultation of CPUID family/model/stepping is not necessary); such features are identified explicitly where they are documented in this manual. Bit 31: Enumerates support for Speculative Store Bypass Disable (SSBD). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[2] (SSBD). **NOTE:** * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.
Structured Extended Feature Enumeration Sub-leaf (Initial EAX Value = 07H, ECX = 1)		
07H		**NOTES:** Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	This field reports 0 if the sub-leaf index, *1*, is invalid. Bit 00: SHA512. If 1, supports the SHA512 instructions. Bit 01: SM3. If 1, supports the SM3 instructions. Bit 02: SM4. If 1, supports the SM4 instructions. Bit 03: Reserved. Bit 04: AVX-VNNI. AVX (VEX-encoded) versions of the Vector Neural Network Instructions. Bit 05: AVX512-BF16. Vector Neural Network Instructions supporting BFLOAT16 inputs and conversion instructions from IEEE single precision. Bit 06: LASS. If 1, supports Linear Address Space Separation. Bit 07: CMPCCXADD. If 1, supports the CMPccXADD instruction. Bit 08: ArchPerfMonExt. If 1, supports ArchPerfMonExt. When set, indicates that the Architectural Performance Monitoring Extended Leaf (EAX = 23H) is valid. Bit 09: Reserved. Bit 10: If 1, supports fast zero-length REP MOVSB. Bit 11: If 1, supports fast short REP STOSB. Bit 12: If 1, supports fast short REP CMPSB, REP SCASB.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Bits 18-13: Reserved. Bit 19: WRMSRNS. If 1, supports the WRMSRNS instruction. Bit 20: Reserved. Bit 21: AMX-FP16. If 1, the processor supports tile computational operations on FP16 numbers. Bit 22: HRESET. If 1, supports history reset via the HRESET instruction and the IA32_HRESET_ENABLE MSR. When set, indicates that the Processor History Reset Leaf (EAX = 20H) is valid. Bit 23: AVX-IFMA. If 1, supports the AVX-IFMA instructions. Bits 25-24: Reserved. Bit 26: LAM. If 1, supports Linear Address Masking. Bit 27: MSRLIST. If 1, supports the RDMSRLIST and WRMSRLIST instructions and the IA32_BARRIER MSR. Bits 29-28: Reserved. Bit 30: INVD_DISABLE_POST_BIOS_DONE. If 1, supports INVD execution prevention after BIOS Done. Bit 31: Reserved.
	EBX	This field reports 0 if the sub-leaf index, *1*, is invalid. Bit 00: Enumerates the presence of the IA32_PPIN and IA32_PPIN_CTL MSRs. If 1, these MSRs are supported. Bits 02-01: Reserved. Bit 03: CPUIDMAXVAL_LIM_RMV. If 1, IA32_MISC_ENABLE[bit 22] cannot be set to 1 to limit the value returned by CPUID.00H:EAX[bits 7:0]. Bits 31-04: Reserved.
	ECX	This field reports 0 if the sub-leaf index, *1*, is invalid; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *1*, is invalid. Bits 03-00: Reserved. Bit 04: AVX-VNNI-INT8. If 1, supports the AVX-VNNI-INT8 instructions. Bit 05: AVX-NE-CONVERT. If 1, supports the AVX-NE-CONVERT instructions. Bits 09-06: Reserved. Bit 10: AVX-VNNI-INT16. If 1, supports the AVX-VNNI-INT16 instructions. Bits 13-11: Reserved. Bit 14: PREFETCHI. If 1, supports the PREFETCHIT0/1 instructions. Bits 16-15: Reserved. Bit 17: UIRET_UIF. If 1, UIRET sets UIF to the value of bit 1 of the RFLAGS image loaded from the stack. Bit 18: CET_SSS. If 1, indicates that an operating system can enable supervisor shadow stacks as long as it ensures that a supervisor shadow stack cannot become prematurely busy due to page faults (see Section 18.2.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). When emulating the CPUID instruction, a virtual-machine monitor (VMM) should return this bit as 1 only if it ensures that VM exits cannot cause a guest supervisor shadow stack to appear to be prematurely busy. Such a VMM could set the "prematurely busy shadow stack" VM-exit control and use the additional information that it provides. Bit 19: AVX10. If 1, supports the Intel® AVX10 instructions and indicates the presence of CPUID Leaf 24H, which enumerates version number and supported vector lengths. Bits 31-20: Reserved.
	Structured Extended Feature Enumeration Sub-leaf (Initial EAX Value = 07H, ECX = 2)	
07H		**NOTES:** Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved.
	EBX	This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved.
	ECX	This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	This field reports 0 if the sub-leaf index, 2, is invalid. Bit 00: PSFD. If 1, indicates bit 7 of the IA32_SPEC_CTRL MSR is supported. Bit 7 of this MSR disables Fast Store Forwarding Predictor without disabling Speculative Store Bypass. Bit 01: IPRED_CTRL. If 1, indicates bits 3 and 4 of the IA32_SPEC_CTRL MSR are supported. Bit 3 of this MSR enables IPRED_DIS control for CPL3. Bit 4 of this MSR enables IPRED_DIS control for CPL0/1/2. Bit 02: RRSBA_CTRL. If 1, indicates bits 5 and 6 of the IA32_SPEC_CTRL MSR are supported. Bit 5 of this MSR disables RRSBA behavior for CPL3. Bit 6 of this MSR disables RRSBA behavior for CPL0/1/2. Bit 03: DDPD_U. If 1, indicates bit 8 of the IA32_SPEC_CTRL MSR is supported. Bit 8 of this MSR disables Data Dependent Prefetcher. Bit 04: BHI_CTRL. If 1, indicates bit 10 of the IA32_SPEC_CTRL MSR is supported. Bit 10 of this MSR enables BHI_DIS_S behavior. Bit 05: MCDT_NO. Processors that enumerate this bit as 1 do not exhibit MXCSR Configuration Dependent Timing (MCDT) behavior and do not need to be mitigated to avoid data-dependent behavior for certain instructions. Bit 06: If 1, supports the UC-lock disable feature and it causes #AC. Bit 07: MONITOR_MITG_NO. If 1, indicates that the MONITOR/UMONITOR instructions are not affected by performance or power issues due to MONITOR/UMONITOR instructions exceeding the capacity of an internal monitor tracking table. If 0, then the product may be affected by this issue. Bits 31-08: Reserved.
	Direct Cache Access Information Leaf (Initial EAX Value = 09H)	
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
	Architectural Performance Monitoring Leaf (Initial EAX Value = 0AH)	
0AH	EAX	Bits 07-00: Version ID of architectural performance monitoring. Bits 15-08: Number of general-purpose performance monitoring counter per logical processor. Bits 23-16: Bit width of general-purpose, performance monitoring counter. Bits 31-24: Length of EBX bit vector to enumerate architectural performance monitoring events. Architectural event x is supported if EBX[x]=0 && EAX[31:24]>x.
	EBX	Bit 00: Core cycle event not available if 1 or if EAX[31:24]<1. Bit 01: Instruction retired event not available if 1 or if EAX[31:24]<2. Bit 02: Reference cycles event not available if 1 or if EAX[31:24]<3. Bit 03: Last-level cache reference event not available if 1 or if EAX[31:24]<4. Bit 04: Last-level cache misses event not available if 1 or if EAX[31:24]<5. Bit 05: Branch instruction retired event not available if 1 or if EAX[31:24]<6. Bit 06: Branch mispredict retired event not available if 1 or if EAX[31:24]<7. Bit 07: Top-down slots event not available if 1 or if EAX[31:24]<8. Bits 31-08: Reserved = 0.
	ECX	Bits 31-00: Supported fixed counters bit mask. Fixed-function performance counter 'i' is supported if bit 'i' is 1 (first counter index starts at zero). It is recommended to use the following logic to determine if a Fixed Counter is supported: FxCtr[i]_is_supported := ECX[i] || (EDX[4:0] > i); [1]
	EDX	Bits 04-00: Number of contiguous fixed-function performance counters starting from 0 (if Version ID > 1).[1] Bits 12-05: Bit width of fixed-function performance counters (if Version ID > 1). Bits 14-13: Reserved = 0. Bit 15: AnyThread deprecation. Bits 31-16: Reserved = 0.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Extended Topology Enumeration Leaf (Initial EAX Value = 0BH, ECX ≥ 0)
0BH		**NOTES:** *CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.* The sub-leaves of CPUID leaf 0BH describe an ordered hierarchy of logical processors starting from the smallest-scoped domain of a Logical Processor (sub-leaf index 0) to the Core domain (sub-leaf index 1) to the largest-scoped domain (the last valid sub-leaf index) that is implicitly subordinate to the unenumerated highest-scoped domain of the processor package (socket). The details of each valid domain is enumerated by a corresponding sub-leaf. Details for a domain include its type and how all instances of that domain determine the number of logical processors and x2 APIC ID partitioning at the next higher-scoped domain. The ordering of domains within the hierarchy is fixed architecturally as shown below. For a given processor, not all domains may be relevant or enumerated; however, the logical processor and core domains are always enumerated. For two valid sub-leaves N and N+1, sub-leaf N+1 represents the next immediate higher-scoped domain with respect to the domain of sub-leaf N for the given processor. If sub-leaf index "N" returns an invalid domain type in ECX[15:08] (00H), then all sub-leaves with an index greater than "N" shall also return an invalid domain type. A sub-leaf returning an invalid domain always returns 0 in EAX and EBX.
	EAX	Bits 04-00: The number of bits that the x2APIC ID must be shifted to the right to address instances of the next higher-scoped domain. When logical processor is not supported by the processor, the value of this field at the Logical Processor domain sub-leaf may be returned as either 0 (no allocated bits in the x2APIC ID) or 1 (one allocated bit in the x2APIC ID); software should plan accordingly. Bits 31-05: Reserved.
	EBX	Bits 15-00: The number of logical processors across all instances of this domain within the next higher-scoped domain. (For example, in a processor socket/package comprising "M" dies of "N" cores each, where each core has "L" logical processors, the "die" domain sub-leaf value of this field would be M*N*L.) This number reflects configuration as shipped by Intel. Note, software must not use this field to enumerate processor topology*. Bits 31-16: Reserved.
	ECX	Bits 07-00: The input ECX sub-leaf index. Bits 15-08: Domain Type. This field provides an identification value which indicates the domain as shown below. Although domains are ordered, their assigned identification values are not and software should not depend on it. <u>Hierarchy</u> / <u>Domain</u> / <u>Domain Type Identification Value</u> Lowest / Logical Processor / 1 Highest / Core / 2 (Note that enumeration values of 0 and 3-255 are reserved.) Bits 31-16: Reserved.
	EDX	Bits 31-00: x2APIC ID of the current logical processor. **NOTES:** * Software must not use the value of EBX[15:0] to enumerate processor topology of the system. The value is only intended for display and diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.
		Processor Extended State Enumeration Main Leaf (Initial EAX Value = 0DH, ECX = 0)
0DH		**NOTES:** Leaf 0DH main leaf (ECX = 0).

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
	EAX	Bits 31-00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1. Bit 00: x87 state. Bit 01: SSE state. Bit 02: AVX state. Bits 04-03: MPX state. Bits 07-05: AVX-512 state. Bit 08: Used for IA32_XSS. Bit 09: PKRU state. Bits 16-10: Used for IA32_XSS. Bit 17: TILECFG state. Bit 18: TILEDATA state. Bits 31-19: Reserved.
	EBX	Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0.
	EDX	Bit 31-00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31-00: Reserved.
	Processor Extended State Enumeration Sub-leaf (Initial EAX Value = 0DH, ECX = 1)	
0DH	EAX	Bit 00: XSAVEOPT is available. Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set. Bit 02: Supports XGETBV with ECX = 1 if set. Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set. Bit 04: Supports extended feature disable (XFD) if set. Bits 31-05: Reserved.
	EBX	Bits 31-00: The size in bytes of the XSAVE area containing all states enabled by XCRO | IA32_XSS. **NOTES:** If EAX[3] is enumerated as 0 and EAX[1] is enumerated as 1, EBX enumerates the size of the XSAVE area containing all states enabled by XCRO. If EAX[1] and EAX[3] are both enumerated as 0, EBX enumerates zero.
	ECX	Bits 31-00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07-00: Used for XCR0. Bit 08: PT state. Bit 09: Used for XCR0. Bit 10: PASID state. Bit 11: CET user state. Bit 12: CET supervisor state. Bit 13: HDC state. Bit 14: UINTR state. Bit 15: LBR state (only for the architectural LBR feature). Bit 16: HWP state. Bits 18-17: Used for XCR0. Bits 31-19: Reserved.
	EDX	Bits 31-00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31-00: Reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Processor Extended State Enumeration Sub-leaves (Initial EAX Value = 0DH, ECX = n, n > 1)
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (0 ≤ n ≤ 31) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (32 ≤ n ≤ 63) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].
	EAX	Bits 31-00: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*.
	EBX	Bits 31-00: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCR0 register*.
	ECX	Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCR0. Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31-02 are reserved. This field reports 0 if the sub-leaf index, n, is invalid*.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
		Intel® Resource Director Technology (Intel® RDT) Monitoring Enumeration Sub-leaf (Initial EAX Value = 0FH, ECX = 0)
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.
	EAX	Reserved.
	EBX	Bits 31-00: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31-02: Reserved.
		L3 Cache Intel® RDT Monitoring Capability Enumeration Sub-leaf (Initial EAX Value = 0FH, ECX = 1)
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX.
	EAX	Bits 07-00:The counter width is encoded as an offset from 24b. A value of zero in this field indicates that 24-bit counters are supported. A value of 8 in this field indicates that 32-bit counters are supported. Bit 08: If 1, indicates the presence of an overflow bit in the IA32_QM_CTR MSR (bit 61). Bit 09: If 1, indicates the presence of non-CPU agent Intel RDT CMT support. Bit 10: If 1, indicates the presence of non-CPU agent Intel RDT MBM support. Bits 31-11: Reserved.
	EBX	Bits 31-00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes) and Memory Bandwidth Monitoring (MBM) metrics.
	ECX	Maximum range (zero-based) of RMID of this resource type.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31-03: Reserved.
	Intel® Resource Director Technology (Intel® RDT) Allocation Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = 0)	
10H		**NOTES:** Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.
	EAX	Reserved.
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31-04: Reserved.
	ECX	Reserved.
	EDX	Reserved.
	L3 Cache Allocation Technology Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = ResID =1)	
10H		**NOTES:** Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04-00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31-05: Reserved.
	EBX	Bits 31-00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bit 00: Reserved. Bit 01: If 1, indicates L3 CAT for non-CPU agents is supported. Bit 02: If 1, indicates L3 Code and Data Prioritization Technology is supported. Bit 03: If 1, indicates non-contiguous capacity bitmask is supported. The bits that are set in the various IA32_L3_MASK_n registers do not have to be contiguous. Bits 31-04: Reserved.
	EDX	Bits 15-00: Highest Class of Service (CLOS) number supported for this ResID. Bits 31-16: Reserved.
	L2 Cache Allocation Technology Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = ResID =2)	
10H		**NOTES:** Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04-00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31-05: Reserved.
	EBX	Bits 31-00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bits 01-00: Reserved. Bit 02: CDP. If 1, indicates L2 Code and Data Prioritization Technology is supported. Bit 03: If 1, indicates non-contiguous capacity bitmask is supported. The bits that are set in the various IA32_L2_MASK_n registers do not have to be contiguous. Bits 31-04: Reserved.
	EDX	Bits 15-00: Highest CLOS number supported for this ResID. Bits 31-16: Reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Memory Bandwidth Allocation Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = ResID =3)
10H		**NOTES:** Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 11-00: Reports the maximum MBA throttling value supported for the corresponding ResID. Add one to the return value to get the result. Bits 31-12: Reserved.
	EBX	Bits 31-00: Reserved.
	ECX	Bits 01-00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31-03: Reserved.
	EDX	Bits 15-00: Highest CLOS number supported for this ResID. Bits 31-16: Reserved.
		Intel® SGX Capability Enumeration Leaf, Sub-leaf 0 (Initial EAX Value = 12H, ECX = 0)
12H		**NOTES:** Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bits 04-02: Reserved. Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVIRTCHILD, EDECVIRTCHILD, and ESETCONTEXT. Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC. Bit 07: If 1, indicates Intel SGX supports ENCLU instruction leaf EVERIFYREPORT2. Bits 09-08: Reserved. Bit 10: If 1, indicates Intel SGX supports ENCLS instruction leaf EUPDATESVN. Bit 11: If 1, indicates Intel SGX supports ENCLU instruction leaf EDECCSSA. Bits 31-12: Reserved.
	EBX	Bits 31-00: MISCSELECT. Bit vector of supported extended SGX features.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 07-00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is 2^(EDX[7:0]). Bits 15-08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is 2^(EDX[15:8]). Bits 31-16: Reserved.
		Intel SGX Attributes Enumeration Leaf, Sub-leaf 1 (Initial EAX Value = 12H, ECX = 1)
12H		**NOTES:** Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.
	EBX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.
	ECX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.
	EDX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.
		Intel® SGX EPC Enumeration Leaf, Sub-leaves (Initial EAX Value = 12H, ECX = 2 or higher)
12H		**NOTES:** Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
	EAX	Bit 03-00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.
	Type	0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.
	Type	0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows. EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section. EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved. ECX[03:00]: EPC section property encoding defined as follows: If ECX[3:0] = 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If ECX[3:0] = 0001b, then this section has confidentiality, integrity, and replay protection. If ECX[3:0] = 0010b, then this section has confidentiality protection only. If ECX[3:0] = 0011b, then this section has confidentiality and integrity protection. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.
	Intel® Processor Trace Enumeration Main Leaf (Initial EAX Value = 14H, ECX = 0)	
14H		**NOTES:** Leaf 14H main leaf (ECX = 0).
	EAX	Bits 31-00: Reports the maximum sub-leaf supported in leaf 14H.
	EBX	Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode. Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset. Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets. Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. Bit 06: If 1, indicates support for PSB and PMI preservation. Writes can set IA32_RTIT_CTL[56] (InjectPsb-PmiOnEnable), enabling the processor to set IA32_RTIT_STATUS[7] (PendTopaPMI) and/or IA32_R-TIT_STATUS[6] (PendPSB) in order to preserve ToPA PMIs and/or PSBs otherwise lost due to Intel PT disable. Writes can also set PendToPAPMI and PendPSB. Bit 07: If 1, writes can set IA32_RTIT_CTL[31] (EventEn), enabling Event Trace packet generation. Bit 08: If 1, writes can set IA32_RTIT_CTL[55] (DisTNT), disabling TNT packet generation. Bit 31-09: Reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 02: If 1, indicates support of Single-Range Output scheme. Bit 03: If 1, indicates support of output to Trace Transport subsystem. Bit 30-04: Reserved. Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.
	EDX	Bits 31-00: Reserved.
	Intel® Processor Trace Enumeration Sub-leaf (Initial EAX Value = 14H, ECX = 1)	
14H	EAX	Bits 02-00: Number of configurable Address Ranges for filtering. Bits 15-03: Reserved. Bits 31-16: Bitmap of supported MTC period encodings.
	EBX	Bits 15-00: Bitmap of supported Cycle Threshold value encodings. Bit 31-16: Bitmap of supported Configurable PSB frequency encodings.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
	Time Stamp Counter and Nominal Core Crystal Clock Information Leaf (Initial EAX Value = 15H)	
15H		**NOTES:** If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated. EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency. If ECX is 0, the nominal core crystal clock frequency is not enumerated. "TSC frequency" = "core crystal clock frequency" * EBX/EAX. The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.
	EAX	Bits 31-00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX	Bits 31-00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX	Bits 31-00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX	Bits 31-00: Reserved = 0.
	Processor Frequency Information Leaf (Initial EAX Value = 16H)	
16H	EAX	Bits 15-00: Processor Base Frequency (in MHz). Bits 31-16: Reserved =0.
	EBX	Bits 15-00: Maximum Frequency (in MHz). Bits 31-16: Reserved = 0.
	ECX	Bits 15-00: Bus (Reference) Frequency (in MHz). Bits 31-16: Reserved = 0.
	EDX	Reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		NOTES: * Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces. While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.
		System-On-Chip Vendor Attribute Enumeration Main Leaf (Initial EAX Value = 17H, ECX = 0)
17H		**NOTES:** Leaf 17H main leaf (ECX = 0). Leaf 17H output depends on the initial value in ECX. Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String. Leaf 17H is valid if MaxSOCID_Index >= 3. Leaf 17H sub-leaves 4 and above are reserved.
	EAX	Bits 31-00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.
	EBX	Bits 15-00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31-17: Reserved = 0.
	ECX	Bits 31-00: Project ID. A unique number an SOC vendor assigns to its SOC projects.
	EDX	Bits 31-00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.
		System-On-Chip Vendor Attribute Enumeration Sub-leaf (Initial EAX Value = 17H, ECX = 1..3)
17H	EAX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
	EBX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
	ECX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
	EDX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
		NOTES: Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.
		System-On-Chip Vendor Attribute Enumeration Sub-leaves (Initial EAX Value = 17H, ECX > MaxSOCID_Index)
17H		**NOTES:** Leaf 17H output depends on the initial value in ECX.
	EAX	Bits 31-00: Reserved = 0.
	EBX	Bits 31-00: Reserved = 0.
	ECX	Bits 31-00: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Deterministic Address Translation Parameters Main Leaf (Initial EAX Value = 18H, ECX = 0)
18H		**NOTES:** Each sub-leaf enumerates a different address translation structure. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). See the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product. ** Add one to the return value to get the result.
	EAX	Bits 31-00: Reports the maximum input value of supported sub-leaf in leaf 18H.
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07-04: Reserved. Bits 10-08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15-11: Reserved. Bits 31-16: W = Ways of associativity.
	ECX	Bits 31-00: S = Number of Sets.
	EDX	Bits 04-00: Translation cache type field. 00000b: Null (indicates this sub-leaf is not valid). 00001b: Data TLB. 00010b: Instruction TLB. 00011b: Unified TLB*. 00100b: Load Only TLB. Hit on loads; fills on both loads and stores. 00101b: Store Only TLB. Hit on stores; fill on stores. All other encodings are reserved. Bits 07-05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13-09: Reserved. Bits 25-14: Maximum number of addressable IDs for logical processors sharing this translation cache.** Bits 31-26: Reserved.
		Deterministic Address Translation Parameters Sub-leaf (Initial EAX Value = 18H, ECX ≥ 1)
18H		**NOTES:** Each sub-leaf enumerates a different address translation structure. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch. See the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product. ** Add one to the return value to get the result.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EAX	Bits 31-00: Reserved.
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07-04: Reserved. Bits 10-08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15-11: Reserved. Bits 31-16: W = Ways of associativity.
	ECX	Bits 31-00: S = Number of Sets.
	EDX	Bits 04-00: Translation cache type field. 0000b: Null (indicates this sub-leaf is not valid). 0001b: Data TLB. 0010b: Instruction TLB. 0011b: Unified TLB*. All other encodings are reserved. Bits 07-05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13-09: Reserved. Bits 25-14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31-26: Reserved.
	Key Locker Leaf (Initial EAX Value = 19H)	
19H	EAX	Bit 00: Key Locker restriction of CPL0-only supported. Bit 01: Key Locker restriction of no-encrypt supported. Bit 02: Key Locker restriction of no-decrypt supported. Bits 31-03: Reserved.
	EBX	Bit 00: AESKLE. If 1, the AES Key Locker instructions are fully enabled. Bit 01: Reserved. Bit 02: If 1, the AES wide Key Locker instructions are supported. Bit 03: Reserved. Bit 04: If 1, the platform supports the Key Locker MSRs (IA32_COPY_LOCAL_TO_PLATFORM, IA23_COPY_PLATFORM_TO_LOCAL, IA32_COPY_STATUS, and IA32_IWKEYBACKUP_STATUS) and backing up the internal wrapping key. Bits 31-05: Reserved.
	ECX	Bit 00: If 1, the NoBackup parameter to LOADIWKEY is supported. Bit 01: If 1, KeySource encoding of 1 (randomization of the internal wrapping key) is supported. Bits 31-02: Reserved.
	EDX	Reserved.
	Native Model ID Enumeration Leaf (Initial EAX Value = 1AH, ECX = 0)	
1AH		**NOTES:** This leaf exists on all hybrid parts, however this leaf is not only available on hybrid parts. The following algorithm is used for detection of this leaf: If CPUID.0.MAXLEAF ≥ 1AH and CPUID.1A.EAX ≠ 0, then the leaf exists.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EAX	Enumerates the native model ID and core type. Bits 31-24: Core type* 10H: Reserved 20H: Intel Atom® 30H: Reserved 40H: Intel® Core™ Bits 23-00: Native model ID of the core. The core-type and native model ID can be used to uniquely identify the microarchitecture of the core. This native model ID is not unique across core types, and not related to the model ID reported in CPUID leaf 01H, and does not identify the SOC. * The core type may only be used as an identification of the microarchitecture for this logical processor and its numeric value has no significance, neither large nor small. This field neither implies nor expresses any other attribute to this logical processor and software should not assume any.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
	PCONFIG Information Sub-leaf (Initial EAX Value = 1BH, ECX ≥ 0)	
1BH		For details on this sub-leaf, see “INPUT EAX = 1BH: Returns PCONFIG Information” on page 3-258. **NOTE:** Leaf 1BH is supported if CPUID.(EAX=07H, ECX=0H):EDX[18] = 1.
	Last Branch Records Information Leaf (Initial EAX Value = 1CH)	
1CH		**NOTE:** This leaf pertains to the architectural feature.
	EAX	Bits 07-00: Supported LBR Depth Values. For each bit n set in this field, the IA32_LBR_DEPTH.DEPTH value 8*(n+1) is supported. Bits 29-08: Reserved. Bit 30: Deep C-state Reset. If set, indicates that LBRs may be cleared on an MWAIT that requests a C-state numerically greater than C1. Bit 31: IP Values Contain LIP. If set, LBR IP values contain LIP. If clear, IP values contain Effective IP.
	EBX	Bit 00: CPL Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[2:1] to non-zero value. Bit 01: Branch Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[22:16] to non-zero value. Bit 02: Call-stack Mode Supported. If set, the processor supports setting IA32_LBR_CTL[3] to 1. Bits 31-03: Reserved.
	ECX	Bit 00: Mispredict Bit Supported. IA32_LBR_x_INFO[63] holds indication of branch misprediction (MISPRED). Bit 01: Timed LBRs Supported. IA32_LBR_x_INFO[15:0] holds CPU cycles since last LBR entry (CYC_CNT), and IA32_LBR_x_INFO[60] holds an indication of whether the value held there is valid (CYC_CNT_VALID). Bit 02: Branch Type Field Supported. IA32_LBR_INFO_x[59:56] holds indication of the recorded operation’s branch type (BR_TYPE). Bits 15-03: Reserved. Bits 19-16: Event Logging Supported bitmap. Bits 31-20: Reserved.
	EDX	Bits 31-00: Reserved.
	Tile Information Main Leaf (Initial EAX Value = 1DH, ECX = 0)	
1DH		**NOTES:** For sub-leaves of 1DH, they are indexed by the palette id. Leaf 1DH sub-leaves 2 and above are reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EAX	Bits 31-00: max_palette. Highest numbered palette sub-leaf. Value = 1.
	EBX	Bits 31-00: Reserved = 0.
	ECX	Bits 31-00: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
	Tile Palette 1 Sub-leaf (Initial EAX Value = 1DH, ECX = 1)	
1DH	EAX	Bits 15-00: Palette 1 total_tile_bytes. Value = 8192. Bits 31-16: Palette 1 bytes_per_tile. Value = 1024.
	EBX	Bits 15-00: Palette 1 bytes_per_row. Value = 64. Bits 31-16: Palette 1 max_names (number of tile registers). Value = 8.
	ECX	Bits 15-00: Palette 1 max_rows. Value = 16. Bits 31-16: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
	TMUL Information Main Leaf (Initial EAX Value = 1EH, ECX = 0)	
1EH		**NOTE:** Leaf 1EH sub-leaves 1 and above are reserved.
	EAX	Bits 31-00: Reserved = 0.
	EBX	Bits 07-00: tmul_maxk (rows or columns). Value = 16. Bits 23-08: tmul_maxn (column bytes). Value = 64. Bits 31-24: Reserved = 0.
	ECX	Bits 31-00: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
	V2 Extended Topology Enumeration Leaf (Initial EAX Value = 1FH, ECX ≥ 0)	
1FH		**NOTES:** *CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends using leaf 1FH when available rather than leaf 0BH and ensuring that any leaf 0BH algorithms are updated to support leaf 1FH.* The sub-leaves of CPUID leaf 1FH describe an ordered hierarchy of logical processors starting from the smallest-scoped domain of a Logical Processor (sub-leaf index 0) to the Core domain (sub-leaf index 1) to the largest-scoped domain (the last valid sub-leaf index) that is implicitly subordinate to the unenumerated highest-scoped domain of the processor package (socket). The details of each valid domain is enumerated by a corresponding sub-leaf. Details for a domain include its type and how all instances of that domain determine the number of logical processors and x2 APIC ID partitioning at the next higher-scoped domain. The ordering of domains within the hierarchy is fixed architecturally as shown below. For a given processor, not all domains may be relevant or enumerated; however, the logical processor and core domains are always enumerated. As an example, a processor may report an ordered hierarchy consisting only of "Logical Processor," "Core," and "Die." For two valid sub-leaves N and N+1, sub-leaf N+1 represents the next immediate higher-scoped domain with respect to the domain of sub-leaf N for the given processor. If sub-leaf index "N" returns an invalid domain type in ECX[15:08] (00H), then all sub-leaves with an index greater than "N" shall also return an invalid domain type. A sub-leaf returning an invalid domain always returns 0 in EAX and EBX.
	EAX	Bits 04-00: The number of bits that the x2APIC ID must be shifted to the right to address instances of the next higher-scoped domain. When logical processor is not supported by the processor, the value of this field at the Logical Processor domain sub-leaf may be returned as either 0 (no allocated bits in the x2APIC ID) or 1 (one allocated bit in the x2APIC ID); software should plan accordingly. Bits 31-05: Reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 15-00: The number of logical processors across all instances of this domain within the next higher-scoped domain relative to this current logical processor. (For example, in a processor socket/package comprising "M" dies of "N" cores each, where each core has "L" logical processors, the "die" domain sub-leaf value of this field would be M*N*L. In an asymmetric topology this would be the summation of the value across the lower domain level instances to create each upper domain level instance.) This number reflects configuration as shipped by Intel. Note, software must not use this field to enumerate processor topology*. Bits 31-16: Reserved.
	ECX	Bits 07-00: The input ECX sub-leaf index. Bits 15-08: Domain Type. This field provides an identification value which indicates the domain as shown below. Although domains are ordered, as also shown below, their assigned identification values are not and software should not depend on it. (For example, if a new domain between core and module is specified, it will have an identification value higher than 5.) Hierarchy / Domain / Domain Type Identification Value Lowest / Logical Processor / 1 ... / Core / 2 ... / Module / 3 ... / Tile / 4 ... / Die / 5 ... / DieGrp / 6 Highest / Package/Socket / (implied) (Note that enumeration values of 0 and 7-255 are reserved.) Bits 31-16: Reserved.
	EDX	Bits 31-00: x2APIC ID of the current logical processor. It is always valid and does not vary with the sub-leaf index in ECX. **NOTES:** * Software must not use the value of EBX[15:0] to enumerate processor topology of the system. The value is only intended for display and diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.
	Processor History Reset Sub-leaf (Initial EAX Value = 20H, ECX = 0)	
20H	EAX	Reports the maximum number of sub-leaves that are supported in leaf 20H.
	EBX	Indicates which bits may be set in the IA32_HRESET_ENABLE MSR to enable reset of different components of hardware-maintained history. Bit 00: Indicates support for both HRESET's EAX[0] parameter, and IA32_HRESET_ENABLE[0] set by the OS to enable reset of Intel® Thread Director history. Bits 31-01: Reserved = 0.
	ECX	Reserved.
	EDX	Reserved.
	Architectural Performance Monitoring Extended Main Leaf (Initial EAX Value = 23H, ECX = 0)	
23H		**NOTE:** Output depends on ECX input value.
	EAX	Bits 31-0: If bit *n* is set, sub-leaf *n* is supported. (For unsupported sub-leaves, 0 is returned in the registers EAX, EBX, ECX, and EDX.)

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
	EBX	Bit 00: UnitMask2 supported. If set, the processor supports the UnitMask2 field in the IA32_PERFEVTSELx MSRs. Bit 01: EQ-bit supported. If set, the processor supports the equal flag in the IA32_PERFEVTSELx MSRs. Bits 31-02: Reserved.
	ECX	Bits 07-00: Number of Top-down Microarchitecture Analysis (TMA) slots per cycle. This number can be multiplied by the number of cycles (from CPU_CLK_UNHALTED.THREAD / CPU_CLK_UNHALTED.CORE or IA32_FIXED_CTR1) to determine the total number of slots. Bits 31-08: Reserved.
	EDX	Bits 31-00: Reserved.
Architectural Performance Monitoring Extended Sub-Leaf (Initial EAX Value = 23H, ECX = 1)		
23H	EAX	Bits 31-00: General counters bitmap. For each bit *n* set in this field, the processor supports general-purpose performance monitoring counter *n*.
	EBX	Bits 31-00: Fixed counters bitmap. For each bit *m* set in this field, the processor supports fixed-function performance monitoring counter *m*.[1]
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
Architectural Performance Monitoring Extended Sub-Leaf (Initial EAX Value = 23H, ECX = 2)		
23H	EAX	Bits 31-00: Bitmap of Auto Counter Reload (ACR) general counters that can be reloaded. For each bit *n* set in this field, the processor supports ACR for general-purpose performance monitoring counter *n*.
	EBX	Bits 31-00: Bitmap of Auto Counter Reload (ACR) fixed counters that can be reloaded. For each bit *m* set in this field, the processor supports ACR for fixed-function performance monitoring counter *m*.
	ECX	Bits 31-00: Bitmap of Auto Counter Reload (ACR) general counters that can cause reloads. For each bit *y* set in this field, the processor allows general-purpose performance monitoring counter *y* to reload all existing general-purpose performance monitoring counters capable of being reloaded.
	EDX	Bits 31-00: Bitmap of Auto Counter Reload (ACR) fixed counters that can cause reloads. For each bit *x* set in this field, the processor allows fixed-function performance monitoring counter *x* to reload all existing fixed-function performance monitoring counters capable of being reloaded.
Architectural Performance Monitoring Extended Sub-Leaf (Initial EAX Value = 23H, ECX = 3)		
23H		**NOTE:** Architectural Performance Monitoring Events Bitmap. For each bit *n* set in this field, the processor supports Architectural Performance Monitoring Event of index *n*.
	EAX	Bit 00: Core cycles. Bit 01: Instructions retired. Bit 02: Reference cycles. Bit 03: Last level cache references. Bit 04: Last level cache misses. Bit 05: Branch instructions retired. Bit 06: Branch mispredicts retired. Bit 07: Topdown slots. Bit 08: Topdown backend bound. Bit 09: Topdown bad speculation.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value		Information Provided about the Processor
		Bit 10: Topdown frontend bound. Bit 11: Topdown retiring. Bit 12: LBR inserts. Bits 31-13: Reserved.
	EBX	Bits 31-00: Reserved.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
Converged Vector ISA Main Leaf (Initial EAX Value = 24H, ECX = 0)		
24H		**NOTE:** Output depends on ECX input value.
	EAX	Bits 31-00: Reports the maximum number sub-leaves that are supported in leaf 24H.
	EBX	Bits 07-00: Reports the Intel AVX10 Converged Vector ISA version. Bits 15-08: Reserved. Bit 18-16: Reserved at 111.[2] Bits 31-19: Reserved.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
Unimplemented CPUID Leaf Functions		
21H		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is 21H. If the value returned by CPUID.0:EAX (the maximum input value for basic CPUID information) is at least 21H, 0 is returned in the registers EAX, EBX, ECX, and EDX. Otherwise, the data for the highest basic information leaf is returned.
40000000H — 4FFFFFFFH		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.
Extended Function CPUID Information		
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved.
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode.* Bits 04-01: Reserved. Bit 05: LZCNT. Bits 07-06: Reserved. Bit 08: PREFETCHW. Bits 31-09: Reserved.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 10-00: Reserved. Bit 11: SYSCALL/SYSRET.** Bits 19-12: Reserved = 0. Bit 20: Execute Disable Bit available. Bits 25-21: Reserved = 0. Bit 26: 1-GByte pages are available if 1. Bit 27: RDTSCP and IA32_TSC_AUX are available if 1. Bit 28: Reserved = 0. Bit 29: Intel® 64 Architecture available if 1. Bits 31-30: Reserved = 0. **NOTES:** * LAHF and SAHF are always available in other modes, regardless of the enumeration of this feature flag. ** Intel processors support SYSCALL and SYSRET only in 64-bit mode. This feature flag is always enumerated as 0 outside 64-bit mode.
80000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000004H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000005H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0.
80000006H	EAX EBX	Reserved = 0. Reserved = 0.
	ECX	Bits 07-00: Cache Line size in bytes. Bits 11-08: Reserved. Bits 15-12: L2 Associativity field *. Bits 31-16: Cache size in 1K units.
	EDX	Reserved = 0. **NOTES:** * L2 associativity field encodings: 00H - Disabled; 08H - 16 ways 01H - 1 way (direct mapped); 09H - Reserved 02H - 2 ways; 0AH - 32 ways 03H - Reserved; 0BH - 48 ways 04H - 4 ways; 0CH - 64 ways 05H - Reserved; 0DH - 96 ways 06H - 8 ways; 0EH - 128 ways 07H - See CPUID leaf 04H, sub-leaf 2**; 0FH - Fully associative ** CPUID leaf 04H provides details of deterministic cache parameters, including the L2 cache in sub-leaf 2

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
80000007H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Bits 07-00: Reserved = 0. Bit 08: Invariant TSC available if 1. Bits 31-09: Reserved = 0.
80000008H	EAX	Linear/Physical Address size. Bits 07-00: #Physical Address Bits*. Bits 15-08: #Linear Address Bits. Bits 23-16: #Guest Physical Address Bits. This value applies only to software operating in a virtual machine (Intel processors enumerate this value as zero). When this field is zero, refer to #Physical Address Bits for the number of guest physical address bits. Bits 31-24: Reserved = 0.
	EBX	Bits 08-00: Reserved = 0. Bit 09: WBNOINVD is available if 1. Bits 31-10: Reserved = 0.
	ECX EDX	Reserved = 0. Reserved = 0. **NOTES:** * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field. If TME-MK is enabled, the number of bits that can be used to address physical memory is CPUID.80000008H:EAX[7:0] - IA32_TME_ACTIVATE[35:32].

NOTES:

1. The valid range of fixed-function counters is 0 through 15.
2. Earlier versions of this specification documented these bits as enumerating support for different vector lengths. Processors enumerating Intel® AVX10 support all vector widths.

INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string "GenuineIntel" and is expressed:

```
EBX := 756e6547h (* "Genu", with G in the low eight bits of BL *)
EDX := 49656e69h (* "inel", with i in the low eight bits of DL *)
ECX := 6c65746eh (* "ntel", with n in the low eight bits of CL *)
```

INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-18 for available processor type values. Stepping IDs are provided as needed.

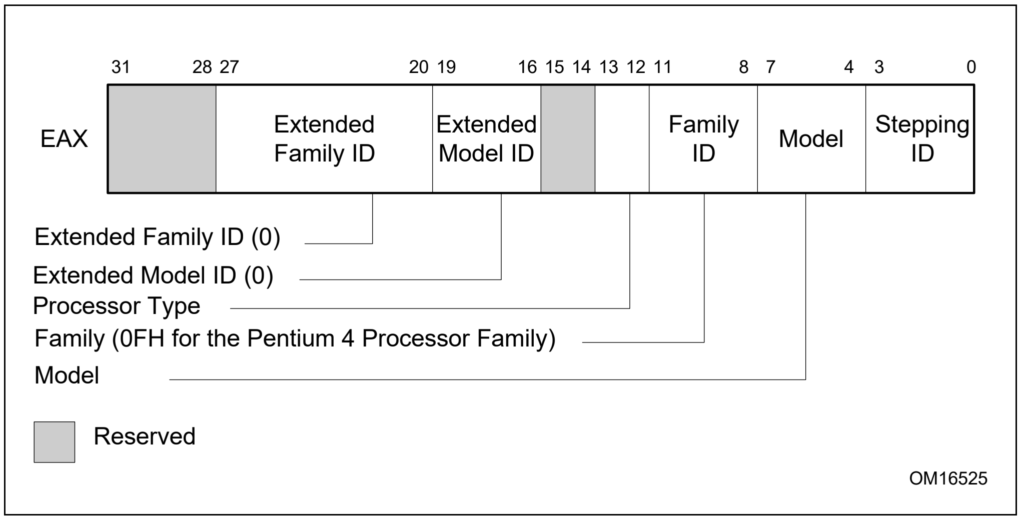


Figure 3-6. Version Information Returned by CPUID in EAX

Table 3-18. Processor Type Field

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

NOTE

See Chapter 21 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN DisplayFamily = Family_ID;
    ELSE DisplayFamily = Extended_Family_ID + Family_ID;
FI;
(* Show DisplayFamily as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN DisplayModel = (Extended_Model_ID « 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field. *)
```

INPUT EAX = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

INPUT EAX = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-19 show encodings for ECX.
- Figure 3-8 and Table 3-20 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

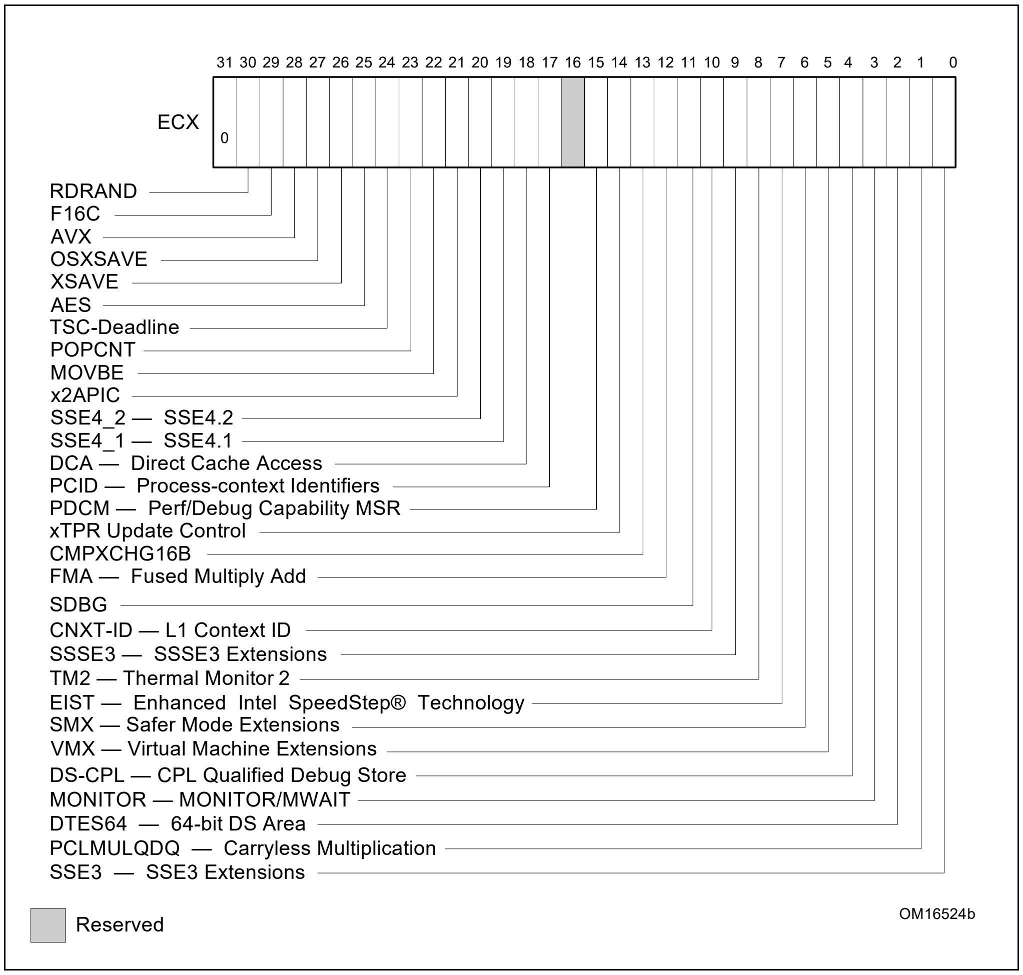


Figure 3-7. Feature Information Returned in the ECX Register

Table 3-19. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3).** A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	**PCLMULQDQ.** A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area.** A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	**Virtual Machine Extensions.** A value of 1 indicates that the processor supports this technology.
6	SMX	**Safer Mode Extensions.** A value of 1 indicates that the processor supports this technology. See Chapter 7, "Safer Mode Extensions Reference."
7	EIST	**Enhanced Intel SpeedStep® technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.
10	CNXT-ID	**L1 Context ID.** A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available.** A value of 1 indicates that the feature is available. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description.
14	xTPR Update Control	**xTPR Update Control.** A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	**PerfMon and Debug Capability:** A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4_1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4_2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.

Table 3-19. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.

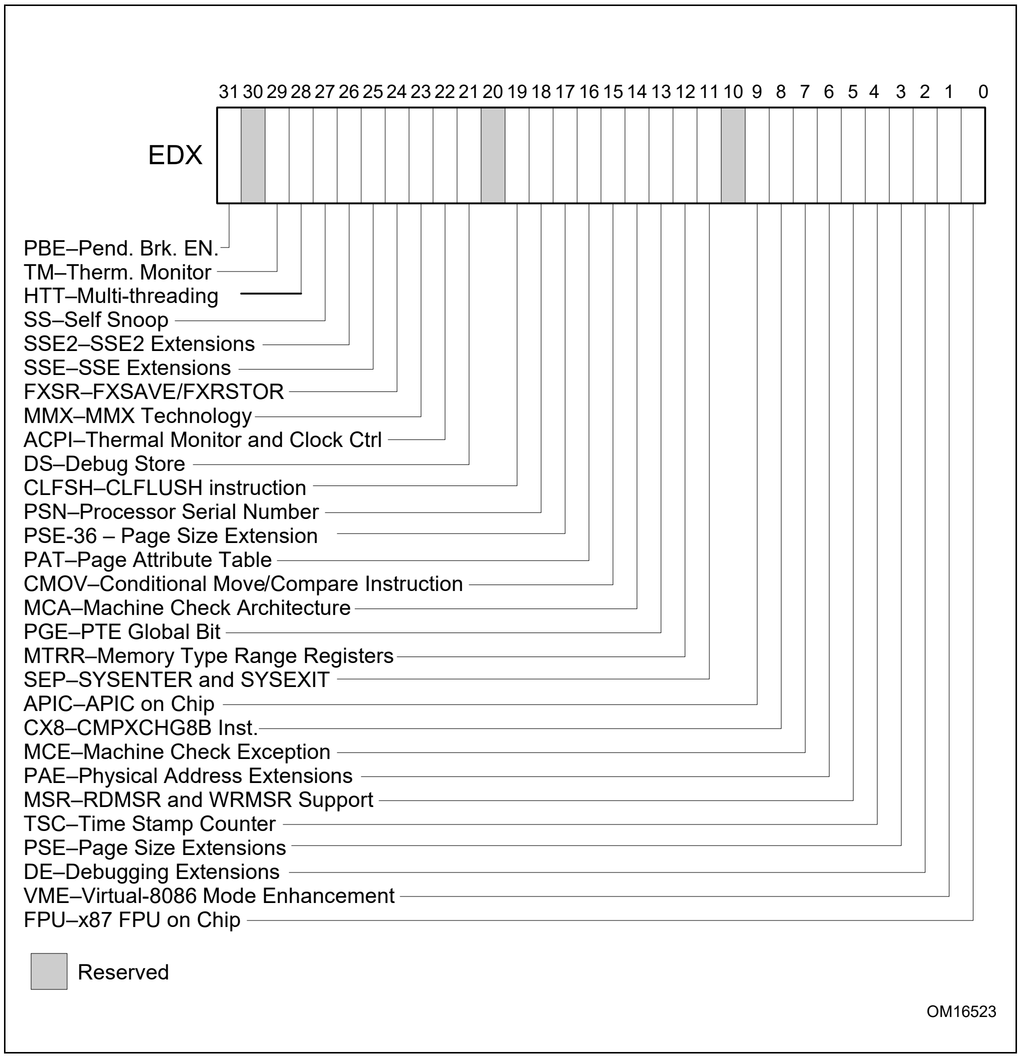


Figure 3-8. Feature Information Returned in the EDX Register

Table 3-20. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	**Floating-Point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	**Machine Check Exception.** Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	**Page Global Bit.** The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	**36-Bit Page Size Extension.** 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	Reserved	Reserved

Table 3-20. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and processor event-based sampling (PEBS) facilities (see Chapter 25, "Introduction to Virtual Machine Extensions," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating-point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Max APIC IDs reserved field is Valid.** A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt.

INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal TLBs, cache, and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-21. Table 3-21 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of "cache type" via CPUID leaf 2.

Table 3-21. Encoding of CPUID Leaf 2 Descriptors

Descriptor Value	Type	Cache or TLB Description
00H	General	Null descriptor, this byte contains no information.
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries.
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries.
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries.
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries.
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries.
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size.
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size.
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size.
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size.
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries.
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size.
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size.
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size.
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size.
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size.
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector.
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector.
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size.
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector.
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector.
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size.
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size.
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache.
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size.
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size.
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size.
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size.
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size.
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size.
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size.
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size.
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size.
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size.
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size.
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size.
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size.
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size.
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries.

Table 3-21. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Descriptor Value	Type	Cache or TLB Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries.
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries.
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries.
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries.
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries.
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries.
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries.
5AH	TLB	Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries.
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries.
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages,128 entries.
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages,256 entries.
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size.
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries.
63H	TLB	Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries.
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 entries.
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size.
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size.
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size.
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries.
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries.
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries.
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries.
70H	Cache	Trace cache: 12 K-μop, 8-way set associative.
71H	Cache	Trace cache: 16 K-μop, 8-way set associative.
72H	Cache	Trace cache: 32 K-μop, 8-way set associative.
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries.
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size.
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector.
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector.
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector.
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector.
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size.
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size.
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size.
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size.
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size.
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size.
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size.
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size.
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size.

Table 3-21. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Descriptor Value	Type	Cache or TLB Description
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries.
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries.
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries.
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries.
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries.
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries.
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries.
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries.
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries.
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries.
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries.
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries.
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GBbyte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 entries.
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries.
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size.
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size.
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size.
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size.
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size.
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size.
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size.
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size.
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size.
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size.
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size.
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size.
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size.
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size.
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size.
F0H	Prefetch	64-Byte prefetching.
F1H	Prefetch	128-Byte prefetching.
FEH	General	CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters.

Example 3-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

EAX	66 5B 50 01H
EBX	0H
ECX	0H
EDX	00 7A 70 00H

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This value should be ignored.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K-μop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-17.

This Cache Size in Bytes

= (Ways + 1) * (Partitions + 1) * (Line_Size + 1) * (Sets + 1)

= (EBX[31:22] + 1) * (EBX[21:12] + 1) * (EBX[11:0] + 1) * (ECX + 1)

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 10, "Multiple-Processor Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-17.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-17.

INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-17.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-17), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-17.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-17) is greater than Pn 0. See Table 3-17.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 25, “Introduction to Virtual Machine Extensions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

INPUT EAX = 0BH: Returns Extended Topology Information

CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-17.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-17.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-17. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
    IF (CPUID.(EAX=0DH, ECX=0H):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
        Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
    FI;
```

INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 0FH and ECX = n (n >= 1, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

INPUT EAX = 12H: Returns Intel SGX Enumeration Information

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-17.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-17.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-17.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-17.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-17.

INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-17.

INPUT EAX = 16H: Returns Processor Frequency Information

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-17.

INPUT EAX = 17H: Returns System-On-Chip Information

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-17.

INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-17.

INPUT EAX = 19H: Returns Key Locker Information

When CPUID executes with EAX set to 19H, the processor returns information about Key Locker. See Table 3-17.

INPUT EAX = 1AH: Returns Native Model ID Information

When CPUID executes with EAX set to 1AH, the processor returns information about Native Model Identification. See Table 3-17.

INPUT EAX = 1BH: Returns PCONFIG Information

When CPUID executes with EAX set to 1BH, the processor returns information about PCONFIG capabilities. This information is enumerated in sub-leaves selected by the value of ECX (starting with 0).

Each sub-leaf of CPUID function 1BH enumerates its **sub-leaf type** in EAX. If a sub-leaf type is 0, the sub-leaf is invalid and zero is returned in EBX, ECX, and EDX. In this case, all subsequent sub-leaves (selected by larger input values of ECX) are also invalid.

The only valid sub-leaf type currently defined is 1, indicating that the sub-leaf enumerates target identifiers for the PCONFIG instruction. Any non-zero value returned in EBX, ECX, or EDX indicates a valid target identifier of the PCONFIG instruction (any value of zero should be ignored). The only target identifier currently defined is 1, indicating TME-MK. See the "PCONFIG—Platform Configuration" instruction in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, for more information.

INPUT EAX = 1CH: Returns Last Branch Record Information

When CPUID executes with EAX set to 1CH, the processor returns information about LBRs (the architectural feature). See Table 3-17.

INPUT EAX = 1DH: Returns Tile Information

When CPUID executes with EAX set to 1DH and ECX = 0H, the processor returns information about tile architecture. See Table 3-17.

When CPUID executes with EAX set to 1DH and ECX = 1H, the processor returns information about tile palette 1. See Table 3-17.

INPUT EAX = 1EH: Returns TMUL Information

When CPUID executes with EAX set to 1EH and ECX = 0H, the processor returns information about TMUL capabilities. See Table 3-17.

INPUT EAX = 1FH: Returns V2 Extended Topology Information

When CPUID executes with EAX set to 1FH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 1FH by verifying (a) the highest leaf index supported by CPUID is >= 1FH, and (b) CPUID.1FH:EBX[15:0] reports a non-zero value. See Table 3-17.

INPUT EAX = 20H: Returns History Reset Information

When CPUID executes with EAX set to 20H, the processor returns information about History Reset. See Table 3-17.

INPUT EAX = 23H: Returns Architectural Performance Monitoring Extended Information

When CPUID executes with EAX set to 23H, the processor returns architectural performance monitoring extended information. See Table 3-17.

INPUT EAX = 24H: Returns Intel AVX10 Converged Vector ISA Information

When CPUID executes with EAX set to 24H, the processor returns Intel AVX10 converged vector ISA information. See Table 3-17.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 21 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

The Processor Brand String Method

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.

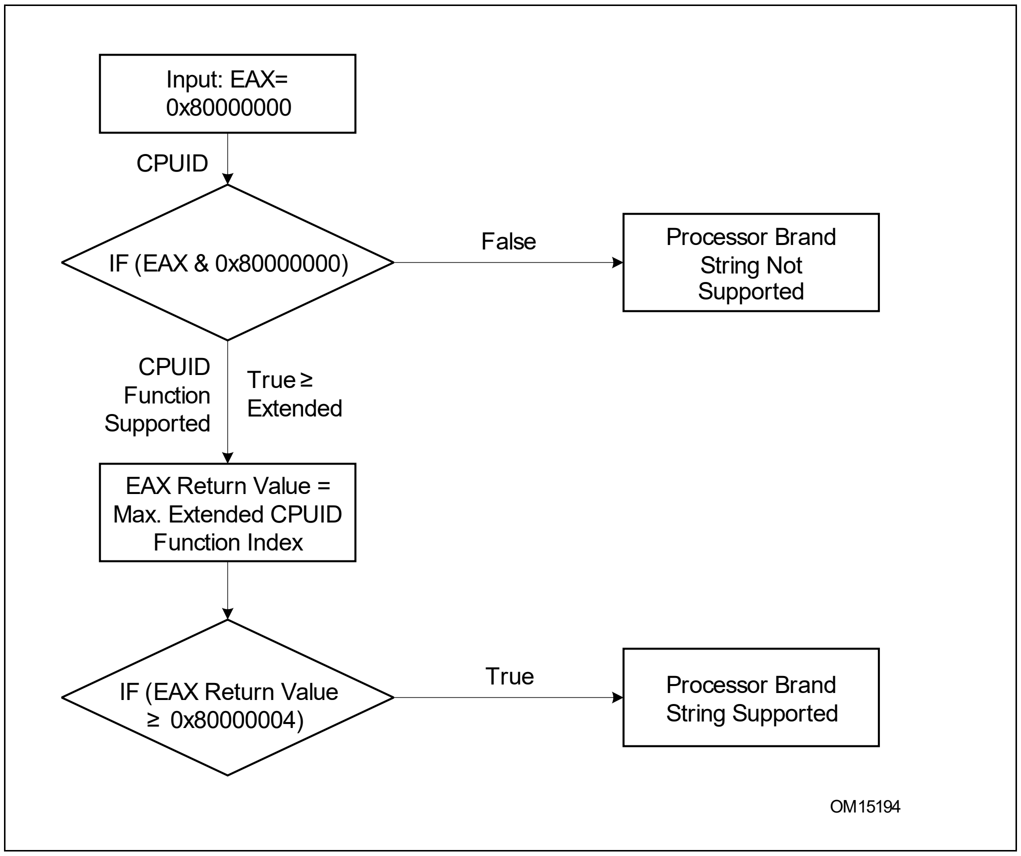

Input: EAX= 0x80000000
CPUID
IF (EAX & 0x80000000)
False
Processor Brand String Not Supported
CPUID Function Supported
True ≥ Extended
EAX Return Value = Max. Extended CPUID Function Index
IF (EAX Return Value ≥ 0x80000004)
True
Processor Brand String Supported
OM15194

Figure 3-9. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-22 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-22. Processor Brand String Returned with Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H	“ ”
	EBX = 20202020H	“ ”
	ECX = 20202020H	“ ”
	EDX = 6E492020H	“nI ”
80000003H	EAX = 286C6574H	“(let”
	EBX = 50202952H	“P)R”
	ECX = 69746E65H	“itne”
	EDX = 52286D75H	“R(mu”

Table 3-22. Processor Brand String Returned with Pentium 4 Processor (Contd.)

EAX Input Value	Return Values	ASCII Equivalent
80000004H	EAX = 20342029H	" 4)"
	EBX = 20555043H	" UPC"
	ECX = 30303531H	"0051"
	EDX = 007A484DH	"\0zHM"

Extracting the Processor Frequency from Brand Strings

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.

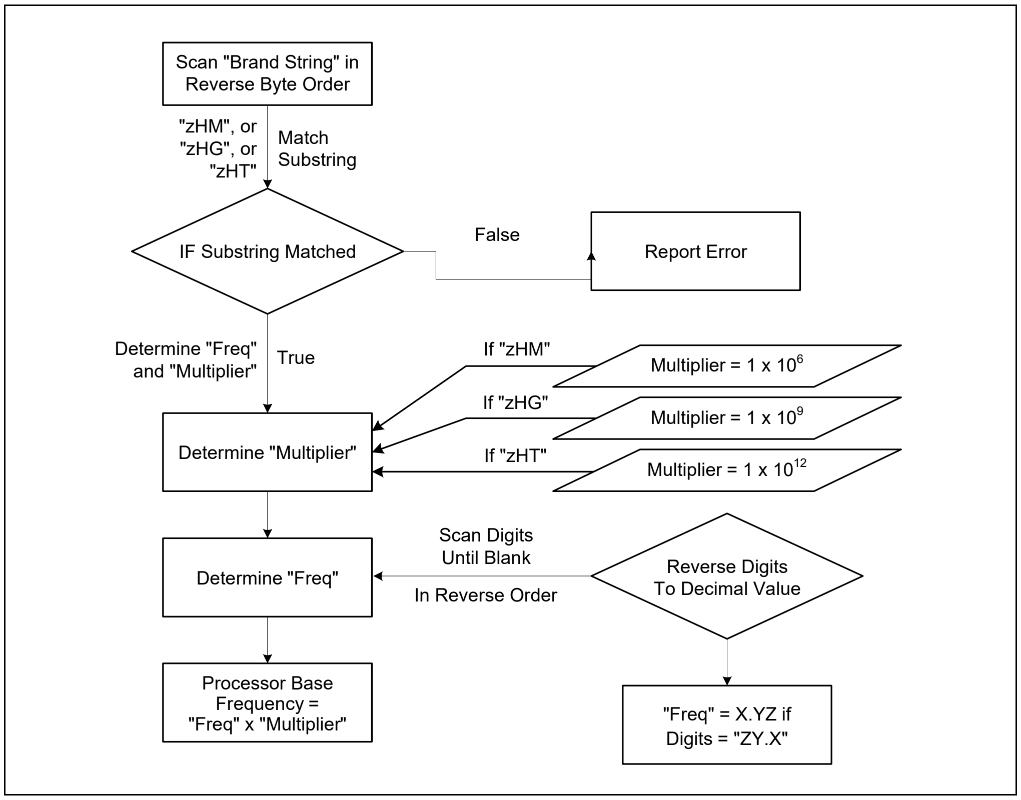


Figure 3-10. Algorithm for Extracting Processor Frequency

The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-23 shows brand indices that have identification strings associated with them.

Table 3-23. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor[1]
02H	Intel(R) Pentium(R) III processor[1]
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor[1]
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor[1]
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor[1]
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor[1]
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor[1]
18H – 0FFH	RESERVED

NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

Operation

```
IA32_BIOS_SIGN_ID MSR := Update with installed microcode revision number;

CASE (EAX) OF
    EAX = 0:
        EAX := Highest basic function input value understood by CPUID;
        EBX := Vendor identification string;
        EDX := Vendor identification string;
        ECX := Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] := Stepping ID;
        EAX[7:4] := Model;
        EAX[11:8] := Family;
        EAX[13:12] := Processor type;
        EAX[15:14] := Reserved;
        EAX[19:16] := Extended Model;
        EAX[27:20] := Extended Family;
        EAX[31:28] := Reserved;
        EBX[7:0] := Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] := CLFLUSH Line Size;
        EBX[16:23] := Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] := Initial APIC ID;
        ECX := Feature flags; (* See Figure 3-7. *)
        EDX := Feature flags; (* See Figure 3-8. *)
    BREAK;
    EAX = 2H:
        EAX := Cache and TLB information;
        EBX := Cache and TLB information;
        ECX := Cache and TLB information;
        EDX := Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX := Reserved;
        EBX := Reserved;
        ECX := ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved. *)
        EDX := ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved. *
    BREAK
    EAX = 4H:
        EAX := Deterministic Cache Parameters Leaf; (* See Table 3-17. *)
        EBX := Deterministic Cache Parameters Leaf;
        ECX := Deterministic Cache Parameters Leaf;
        EDX := Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX := MONITOR/MWAIT Leaf; (* See Table 3-17. *)
        EBX := MONITOR/MWAIT Leaf;
        ECX := MONITOR/MWAIT Leaf;
        EDX := MONITOR/MWAIT Leaf;
    BREAK;
    EAX = 6H:
        EAX := Thermal and Power Management Leaf; (* See Table 3-17. *)
        EBX := Thermal and Power Management Leaf;
        ECX := Thermal and Power Management Leaf;
        EDX := Thermal and Power Management Leaf;
    BREAK;
    EAX = 7H:
        EAX := Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-17. *)
        EBX := Structured Extended Feature Flags Enumeration Leaf;
        ECX := Structured Extended Feature Flags Enumeration Leaf;
        EDX := Structured Extended Feature Flags Enumeration Leaf;
    BREAK;
    EAX = 8H:
        EAX := Reserved = 0;
        EBX := Reserved = 0;
        ECX := Reserved = 0;
        EDX := Reserved = 0;
    BREAK;
    EAX = 9H:
        EAX := Direct Cache Access Information Leaf; (* See Table 3-17. *)
        EBX := Direct Cache Access Information Leaf;
        ECX := Direct Cache Access Information Leaf;
        EDX := Direct Cache Access Information Leaf;
    BREAK;
    EAX = AH:
        EAX := Architectural Performance Monitoring Leaf; (* See Table 3-17. *)
        EBX := Architectural Performance Monitoring Leaf;
        ECX := Architectural Performance Monitoring Leaf;
        EDX := Architectural Performance Monitoring Leaf;
        BREAK
    EAX = BH:
        EAX := Extended Topology Enumeration Leaf; (* See Table 3-17. *)
        EBX := Extended Topology Enumeration Leaf;
        ECX := Extended Topology Enumeration Leaf;
        EDX := Extended Topology Enumeration Leaf;
    BREAK;
    EAX = CH:
        EAX := Reserved = 0;
        EBX := Reserved = 0;
        ECX := Reserved = 0;
        EDX := Reserved = 0;
    BREAK;
    EAX = DH:
        EAX := Processor Extended State Enumeration Leaf; (* See Table 3-17. *)
        EBX := Processor Extended State Enumeration Leaf;
        ECX := Processor Extended State Enumeration Leaf;
        EDX := Processor Extended State Enumeration Leaf;
    BREAK;
    EAX = EH:
        EAX := Reserved = 0;
        EBX := Reserved = 0;
        ECX := Reserved = 0;
        EDX := Reserved = 0;
    BREAK;
    EAX = FH:
        EAX := Intel Resource Director Technology Monitoring Enumeration Leaf; (* See Table 3-17. *)
        EBX := Intel Resource Director Technology Monitoring Enumeration Leaf;
        ECX := Intel Resource Director Technology Monitoring Enumeration Leaf;
        EDX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    BREAK;
    EAX = 10H:
        EAX := Intel Resource Director Technology Allocation Enumeration Leaf; (* See Table 3-17. *)
        EBX := Intel Resource Director Technology Allocation Enumeration Leaf;
        ECX := Intel Resource Director Technology Allocation Enumeration Leaf;
        EDX := Intel Resource Director Technology Allocation Enumeration Leaf;
    BREAK;
    EAX = 12H:
        EAX := Intel SGX Enumeration Leaf; (* See Table 3-17. *)
        EBX := Intel SGX Enumeration Leaf;
        ECX := Intel SGX Enumeration Leaf;
        EDX := Intel SGX Enumeration Leaf;
    BREAK;
    EAX = 14H:
        EAX := Intel Processor Trace Enumeration Leaf; (* See Table 3-17. *)
        EBX := Intel Processor Trace Enumeration Leaf;
        ECX := Intel Processor Trace Enumeration Leaf;
        EDX := Intel Processor Trace Enumeration Leaf;
    BREAK;
    EAX = 15H:
        EAX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (* See Table 3-17. *)
        EBX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
        ECX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
        EDX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
    BREAK;
    EAX = 16H:
        EAX := Processor Frequency Information Enumeration Leaf; (* See Table 3-17. *)
        EBX := Processor Frequency Information Enumeration Leaf;
        ECX := Processor Frequency Information Enumeration Leaf;
        EDX := Processor Frequency Information Enumeration Leaf;
    BREAK;
    EAX = 17H:
        EAX := System-On-Chip Vendor Attribute Enumeration Leaf; (* See Table 3-17. *)
        EBX := System-On-Chip Vendor Attribute Enumeration Leaf;
        ECX := System-On-Chip Vendor Attribute Enumeration Leaf;
        EDX := System-On-Chip Vendor Attribute Enumeration Leaf;
    BREAK;
    EAX = 18H:
        EAX := Deterministic Address Translation Parameters Enumeration Leaf; (* See Table 3-17. *)
        EBX := Deterministic Address Translation Parameters Enumeration Leaf;
        ECX := Deterministic Address Translation Parameters Enumeration Leaf;
        EDX := Deterministic Address Translation Parameters Enumeration Leaf;
    BREAK;
    EAX = 19H:
        EAX := Key Locker Enumeration Leaf; (* See Table 3-17. *)
        EBX := Key Locker Enumeration Leaf;
        ECX := Key Locker Enumeration Leaf;
        EDX := Key Locker Enumeration Leaf;
    BREAK;
    EAX = 1AH:
        EAX := Native Model ID Enumeration Leaf; (* See Table 3-17. *)
        EBX := Native Model ID Enumeration Leaf;
        ECX := Native Model ID Enumeration Leaf;
        EDX := Native Model ID Enumeration Leaf;
    BREAK;
    EAX = 1BH:
        EAX := PCONFIG Information Enumeration Leaf; (* See "INPUT EAX = 1BH: Returns PCONFIG Information" on page 3-258. *)
        EBX := PCONFIG Information Enumeration Leaf;
        ECX := PCONFIG Information Enumeration Leaf;
        EDX := PCONFIG Information Enumeration Leaf;
    BREAK;
    EAX = 1CH:
        EAX := Last Branch Record Information Enumeration Leaf; (* See Table 3-17. *)
        EBX := Last Branch Record Information Enumeration Leaf;
        ECX := Last Branch Record Information Enumeration Leaf;
        EDX := Last Branch Record Information Enumeration Leaf;
    BREAK;
    EAX = 1DH:
        EAX := Tile Information Enumeration Leaf; (* See Table 3-17. *)
        EBX := Tile Information Enumeration Leaf;
        ECX := Tile Information Enumeration Leaf;
        EDX := Tile Information Enumeration Leaf;
    BREAK;
    EAX = 1EH:
        EAX := TMUL Information Enumeration Leaf; (* See Table 3-17. *)
        EBX := TMUL Information Enumeration Leaf;
        ECX := TMUL Information Enumeration Leaf;
        EDX := TMUL Information Enumeration Leaf;
    BREAK;
    EAX = 1FH:
        EAX := V2 Extended Topology Enumeration Leaf; (* See Table 3-17. *)
        EBX := V2 Extended Topology Enumeration Leaf;
        ECX := V2 Extended Topology Enumeration Leaf;
        EDX := V2 Extended Topology Enumeration Leaf;
    BREAK;
    EAX = 20H:
        EAX := Processor History Reset Sub-leaf; (* See Table 3-17. *)
        EBX := Processor History Reset Sub-leaf;
        ECX := Processor History Reset Sub-leaf;
        EDX := Processor History Reset Sub-leaf;
    BREAK;
    EAX = 23H:
        EAX := Architectural Performance Monitoring Extended Leaf; (* See Table 3-17. *)
        EBX := Architectural Performance Monitoring Extended Leaf;
        ECX := Architectural Performance Monitoring Extended Leaf;
        EDX := Architectural Performance Monitoring Extended Leaf;
    BREAK;
    EAX = 24H:
        EAX := Intel AVX10 Converged Vector ISA Leaf; (* See Table 3-17. *)
        EBX := Intel AVX10 Converged Vector ISA Leaf;
        ECX := Intel AVX10 Converged Vector ISA Leaf;
        EDX := Intel AVX10 Converged Vector ISA Leaf;
    BREAK;
    EAX = 80000000H:
        EAX := Highest extended function input value understood by CPUID;
        EBX := Reserved;
        ECX := Reserved;
        EDX := Reserved;
    BREAK;
    EAX = 80000001H:
        EAX := Reserved;
        EBX := Reserved;
        ECX := Extended Feature Bits (* See Table 3-17.*);
        EDX := Extended Feature Bits (* See Table 3-17. *);
    BREAK;
    EAX = 80000002H:
        EAX := Processor Brand String;
        EBX := Processor Brand String, continued;
        ECX := Processor Brand String, continued;
        EDX := Processor Brand String, continued;
    BREAK;
    EAX = 80000003H:
        EAX := Processor Brand String, continued;
        EBX := Processor Brand String, continued;
        ECX := Processor Brand String, continued;
        EDX := Processor Brand String, continued;
    BREAK;
    EAX = 80000004H:
        EAX := Processor Brand String, continued;
        EBX := Processor Brand String, continued;
        ECX := Processor Brand String, continued;
        EDX := Processor Brand String, continued;
    BREAK;
    EAX = 80000005H:
        EAX := Reserved = 0;
        EBX := Reserved = 0;
        ECX := Reserved = 0;
        EDX := Reserved = 0;
    BREAK;
    EAX = 80000006H:
        EAX := Reserved = 0;
        EBX := Reserved = 0;
        ECX := Cache information;
        EDX := Reserved = 0;
    BREAK;
    EAX = 80000007H:
        EAX := Reserved = 0;
        EBX := Reserved = 0;
        ECX := Reserved = 0;
        EDX := Reserved = Misc Feature Flags;
    BREAK;
    EAX = 80000008H:
        EAX := Address Size Information;
        EBX := Misc Feature Flags;
        ECX := Reserved = 0;
        EDX := Reserved = 0;
    BREAK;
    EAX >= 40000000H and EAX <= 4FFFFFFFH:
    DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
        (* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
        EAX := Reserved; (* Information returned for highest basic information leaf. *)
        EBX := Reserved; (* Information returned for highest basic information leaf. *)
        ECX := Reserved; (* Information returned for highest basic information leaf. *)
        EDX := Reserved; (* Information returned for highest basic information leaf. *)
    BREAK;
ESAC;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.
	In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.

CRC32—Accumulate CRC32 Value

Opcode/ Instruction		Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 38 F0 /r	CRC32 r32, r/m8[1]	RM	Valid	Valid	Accumulate CRC32 on r/m8.
F2 0F 38 F1 /r	CRC32 r32, r/m16	RM	Valid	Valid	Accumulate CRC32 on r/m16.
F2 0F 38 F1 /r	CRC32 r32, r/m32	RM	Valid	Valid	Accumulate CRC32 on r/m32.
F2 REX.W 0F 38 F0 /r	CRC32 r64, r/m8	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 REX.W 0F 38 F1 /r	CRC32 r64, r/m64	RM	Valid	N.E.	Accumulate CRC32 on r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

Operation

Notes:

BIT_REFLECT64: DST[63-0] = SRC[0-63]
BIT_REFLECT32: DST[31-0] = SRC[0-31]
BIT_REFLECT16: DST[15-0] = SRC[0-15]
BIT_REFLECT8: DST[7-0] = SRC[0-7]
MOD2: Remainder from Polynomial division modulus 2

CRC32 instruction for 64-bit source operand and 64-bit destination operand:

```
TEMP1[63-0] := BIT_REFLECT64 (SRC[63-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[95-0] := TEMP1[63-0] « 32
TEMP4[95-0] := TEMP2[31-0] « 64
TEMP5[95-0] := TEMP3[95-0] XOR TEMP4[95-0]
TEMP6[31-0] := TEMP5[95-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
DEST[63-32] := 00000000H
```

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

```
TEMP1[31-0] := BIT_REFLECT32 (SRC[31-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[63-0] := TEMP1[31-0] « 32
TEMP4[63-0] := TEMP2[31-0] « 32
TEMP5[63-0] := TEMP3[63-0] XOR TEMP4[63-0]
TEMP6[31-0] := TEMP5[63-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
```

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

```
TEMP1[15-0] := BIT_REFLECT16 (SRC[15-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[47-0] := TEMP1[15-0] « 32
TEMP4[47-0] := TEMP2[31-0] « 16
TEMP5[47-0] := TEMP3[47-0] XOR TEMP4[47-0]
TEMP6[31-0] := TEMP5[47-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
```

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

```
TEMP1[7-0] := BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] := TEMP1[7-0] « 32
TEMP4[39-0] := TEMP2[31-0] « 8
TEMP5[39-0] := TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] := TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
DEST[63-32] := 00000000H
```

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

```
TEMP1[7-0] := BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] := BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] := TEMP1[7-0] « 32
TEMP4[39-0] := TEMP2[31-0] « 8
TEMP5[39-0] := TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] := TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] := BIT_REFLECT (TEMP6[31-0])
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
unsigned int _mm_crc32_u8( unsigned int crc, unsigned char data )
unsigned int _mm_crc32_u16( unsigned int crc, unsigned short data )
unsigned int _mm_crc32_u32( unsigned int crc, unsigned int data )
unsigned __int64 _mm_crc32_u64( unsigned __int64 crc, unsigned __int64 data )
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
	If LOCK prefix is used.

CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert 2 packed signed doubleword integers from xmm2/m64/m32bcst to eight packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512F OR AVX10.1	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double precision floating-point values in the destination operand (the first operand).

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

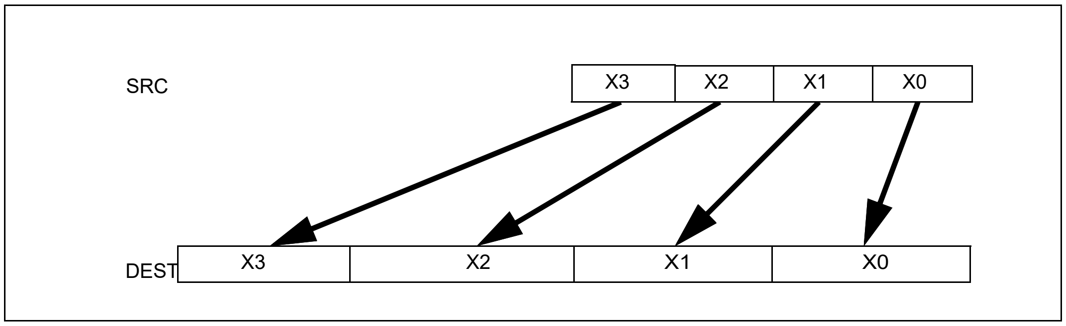


Figure 3-11. CVTDQ2PD (VEX.256 encoded version)

Operation

VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PD (VEX.256 Encoded Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] := 0
```

VCVTDQ2PD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0
```

CVTDQ2PD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PD __m512d _mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d _mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d _mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src);
VCVTDQ2PD __m256d _mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d _mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d _mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m128d _mm_cvtepi32_pd (__m128i src)
```

Other Exceptions

VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-53, “Type E5 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r CVTDQ2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.128.0F.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.256.0F.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single precision floating-point values in ymm1.
EVEX.128.0F.W0 5B /r VCVTDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single precision floating-point values in xmm1with writemask k1.
EVEX.256.0F.W0 5B /r VCVTDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single precision floating-point values in ymm1with writemask k1.
EVEX.512.0F.W0 5B /r VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single precision floating-point values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures
Software Developer's Manual, Volume 1
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures
Software Developer's Manual, Volume 1
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTDQ2PS (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96)
DEST[159:128] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224)
DEST[MAXVL-1:256] := 0
```

VCVTDQ2PS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] := 0
```

CVTDQ2PS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PS __m512 _mm512_cvtepi32_ps( __m512i a);
VCVTDQ2PS __m512 _mm512_mask_cvtepi32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_maskz_cvtepi32_ps( __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_cvt_roundepi32_ps( __m512i a, int r);
VCVTDQ2PS __m512 _mm512_mask_cvt_roundepi_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m512 _mm512_maskz_cvt_roundepi32_ps( __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m256 _mm256_mask_cvtepi32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTDQ2PS __m256 _mm256_maskz_cvtepi32_ps( __mmask8 k, __m256i a);
VCVTDQ2PS __m128 _mm_mask_cvtepi32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTDQ2PS __m128 _mm_maskz_cvtepi32_ps( __mmask8 k, __m128i a);
CVTDQ2PS __m256 _mm256_cvtepi32_ps (__m256i src)
CVTDQ2PS __m128 _mm_cvtepi32_ps (__m128i src)
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.128.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

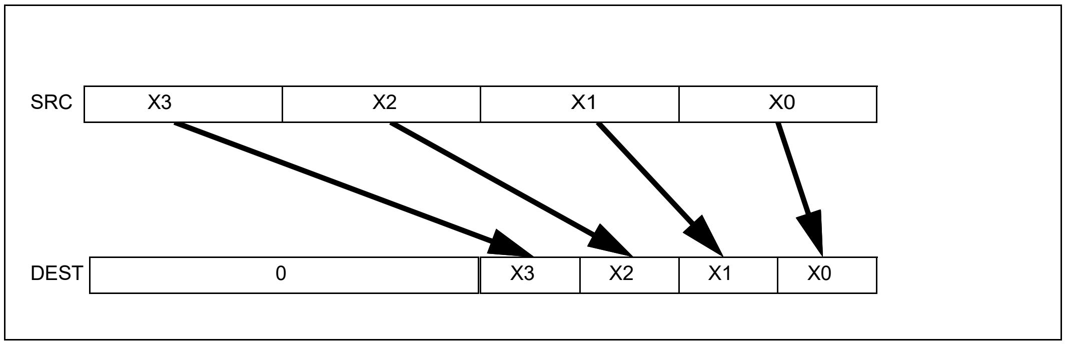


Figure 3-12. VCVTPD2DQ (VEX.256 encoded version)

Operation

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
   i := j * 32
   k := j * 64
   IF k1[j] OR *no writemask*
       THEN DEST[i+31:i] :=
           Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
       ELSE
           IF *merging-masking*                    ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE                                ; zeroing-masking
                   DEST[i+31:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTPD2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTPD2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2DQ __m256i _mm512_cvtpd_epi32(__m512d a);
VCVTPD2DQ __m256i _mm512_mask_cvtpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_maskz_cvtpd_epi32(__mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_cvt_roundpd_epi32(__m512d a, int r);
VCVTPD2DQ __m256i _mm512_mask_cvt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i _mm512_maskz_cvt_roundpd_epi32(__mmask8 k, __m512d a, int r);
VCVTPD2DQ __m128i _mm256_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm256_maskz_cvtpd_epi32(__mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm_maskz_cvtpd_epi32(__mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm256_cvtpd_epi32 (__m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPD2PI—Convert Packed Double Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2D /r CVTPD2PI mm, xmm/m128	RM	V/V	SSE2	Convert two packed double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two packed double precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer32(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTPD1PI __m64 _mm_cvtpd_pi32(__m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 24-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

CVTPD2PS—Convert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four single precision floating-point values in xmm1.
EVEX.128.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two single precision floating-point values in xmm1with writemask k1.
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four single precision floating-point values in xmm1with writemask k1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight single precision floating-point values in ymm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight packed single precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

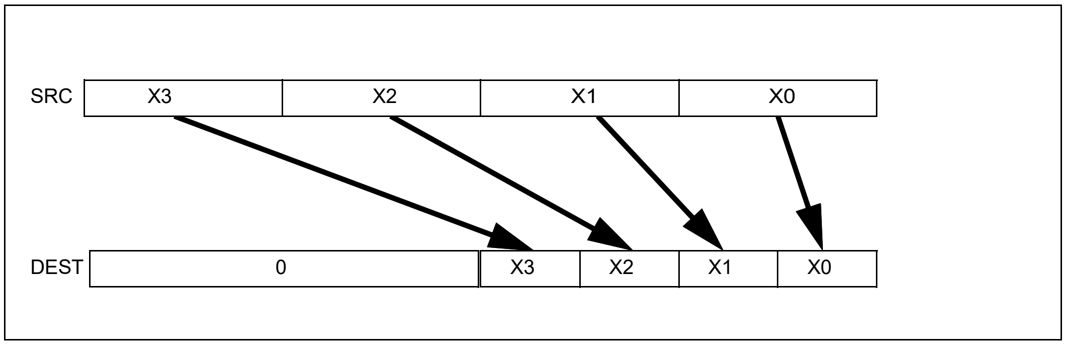


Figure 3-13. VCVTPD2PS (VEX.256 encoded version)

Operation

VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2PS (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTPD2PS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTPD2PS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2PS __m256 _mm512_cvtpd_ps(__m512d a);
VCVTPD2PS __m256 _mm512_mask_cvtpd_ps(__m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_maskz_cvtpd_ps(__mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_cvt_roundpd_ps(__m512d a, int r);
VCVTPD2PS __m256 _mm512_mask_cvt_roundpd_ps(__m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 _mm512_maskz_cvt_roundpd_ps(__mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 _mm256_mask_cvtpd_ps(__m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm256_maskz_cvtpd_ps(__mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm_mask_cvtpd_ps(__m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm_maskz_cvtpd_ps(__mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm256_cvtpd_ps (__m256d a)
CVTPD2PS __m128 _mm_cvtpd_ps (__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPS2DQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.128.66.0F.W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2DQ (Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96)
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])
```

VCVTPS2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] := 0
```

CVTPS2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2DQ __m512i _mm512_cvtps_epi32( __m512 a);
VCVTPS2DQ __m512i _mm512_mask_cvtps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_maskz_cvtps_epi32( __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_cvt_roundps_epi32( __m512 a, int r);
VCVTPS2DQ __m512i _mm512_mask_cvt_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m512i _mm512_maskz_cvt_roundps_epi32( __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m256i _mm256_mask_cvtps_epi32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2DQ __m256i _mm256_maskz_cvtps_epi32( __mmask8 k, __m256 a);
VCVTPS2DQ __m128i _mm_mask_cvtps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2DQ __m128i _mm_maskz_cvtps_epi32( __mmask8 k, __m128 a);
VCVTPS2DQ __ m256i _mm256_cvtps_epi32 (__m256 a)
CVTPS2DQ __m128i _mm_cvtps_epi32 (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPS2PD—Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5A /r CVTPS2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.256.0F.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values in xmm2/m128 to four packed double precision floating-point values in ymm1.
EVEX.128.0F.W0 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed single precision floating-point values in xmm2/m64/m32bcst to packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst {sae}	B	V/V	AVX512F OR AVX10.1	Convert eight packed single precision floating-point values in ymm2/m256/b32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed single precision floating-point values in the source operand (second operand) to two, four or eight packed double precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

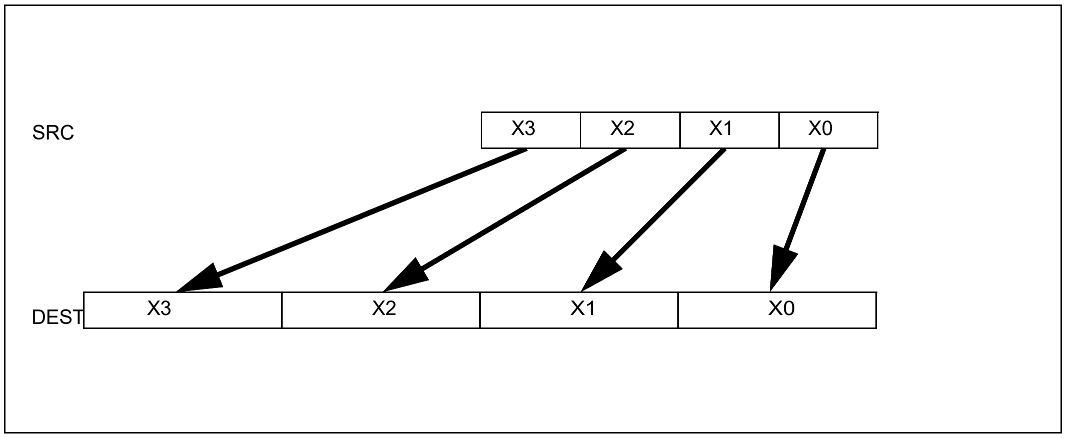


Figure 3-14. CVTPS2PD (VEX.256 encoded version)

Operation

VCVTPS2PD (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
```

```
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2PD (VEX.256 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] := 0
```

VCVTPS2PD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0
```

CVTPS2PD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2PD __m512d _mm512_cvtps_pd( __m256 a);
VCVTPS2PD __m512d _mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTPS2PD __m512d _mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m512d _mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m256d _mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTPS2PD __m128d _mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTPS2PD __m128d _mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_cvtps_pd (__m128 a)
CVTPS2PD __m128d _mm_cvtps_pd (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2D /r CVTPS2PI mm, xmm/m64	RM	Valid	Valid	Convert two packed single precision floating-point values from xmm/m64 to two packed signed doubleword integers in mm.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two packed single precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTPS2PI __m64 _mm_cvtps_pi32(__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 24-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.LIG.F2.0F.W0 2D /r [1] VCVTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.LIG.F2.0F.W1 2D /r [1] VCVTSD2SI r64, xmm1/m64	A	V/N.E.[2]	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LLIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	B	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LLIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SI (EVEX Encoded Version)

```
IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI
```

(V)CVTSD2SI

```
IF 64-Bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTSD2SS—Convert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double precision floating-point value in xmm2/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64	B	V/V	AVX	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2.
EVEX.LLIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a double precision floating-point value in the "convert-from" source operand (the second operand in SSE2 version, otherwise the third operand) to a single precision floating-point value in the destination operand.

When the "convert-from" operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VCVTSD2SS (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

CVTSD2SS (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAXVL-1:32] Unmodified *)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2SS __m128 _mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 _mm_maskz_cvtsd_ss( __mmask8 k, __m128 a,__m128d b);
VCVTSD2SS __m128 _mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_maskz_cvt_roundsd_ss( __mmask8 k, __m128 a,__m128d b, int r);
CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128d b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

CVTSI2SD—Convert Signed Integer to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD xmm1, r32/m32	A	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double precision floating-point value in xmm1.
F2 REX.W 0F 2A /r CVTSI2SD xmm1, r/m64	A	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	C	V/V	AVX512F OR AVX10.1	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	C	V/N.E.[1]	AVX512F OR AVX10.1	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a signed doubleword or quadword integer in the "convert-from" source operand to a double precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VCVTSI2SD (VEX.128 Encoded Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

CVTSI2SD

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d_mm_cvtsi32_sd(__m128d a, int b)
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," if W1; else see Table 2-22, "Type 5 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions," if W1; else see Table 2-61, "Type E10NF Class Exception Conditions."

CVTSI2SS—Convert Signed Integer to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS xmm1, r/m32	A	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64	A	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	C	V/V	AVX512F OR AVX10.1	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	C	V/N.E.[1]	AVX512F OR AVX10.1	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a signed doubleword or quadword integer in the “convert-from” source operand to a single precision floating-point value in the destination operand (first operand). The “convert-from” source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VCVTSI2SS (VEX.128 Encoded Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

CVTSI2SS (128-bit Legacy SSE Version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] :=Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSI2SS __m128 _mm_cvti32_ss(__m128 s, int a);
VCVTSI2SS __m128 _mm_cvt_roundi32_ss(__m128 s, int a, int r);
VCVTSI2SS __m128 _mm_cvti64_ss(__m128 s, __int64 a);
VCVTSI2SS __m128 _mm_cvt_roundi64_ss(__m128 s, __int64 a, int r);
CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

CVTSS2SD—Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single precision floating-point value in xmm2/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2.
EVEX.LLIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a single precision floating-point value in the "convert-from" source operand to a double precision floating-point value in the destination operand. When the "convert-from" source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SD (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN      DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VCVTSS2SD (VEX.128 Encoded Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

CVTSS2SD (128-bit Legacy SSE Version)

```
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b, int r);
VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);
VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b);
VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a,__m128 b);
CVTSS2SD __m128d_mm_cvtss_sd(__m128d a, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.LIG.F3.0F.W0 2D /r [1] VCVTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.LIG.F3.0F.W1 2D /r [1] VCVTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LLIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	B	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LLIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

NOTES:

1. Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed integer in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

(V)CVTSS2SI (Legacy and VEX.128 Encoded Version)

```
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSS2SI int _mm_cvtss_i32( __m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32( __m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64( __m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64( __m128 a, int r);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

CVTTPD2DQ—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.128.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.256.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.128.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W1 E6 /r VCVTTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst {sae}	B	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

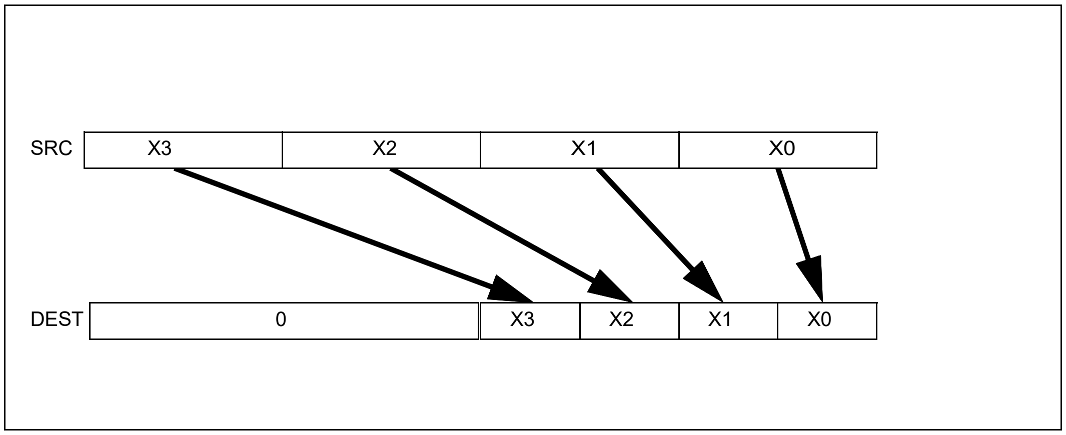


Figure 3-15. VCVTTPD2DQ (VEX.256 encoded version)

Operation

VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
   i := j * 32
   k := j * 64
   IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
   FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTTPD2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192)
DEST[MAXVL-1:128] := 0
```

VCVTTPD2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAXVL-1:64] := 0
```

CVTTPD2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPD2DQ __m256i _mm512_cvttpd_epi32( __m512d a);
VCVTTPD2DQ __m256i _mm512_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_maskz_cvttpd_epi32( __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_cvtt_roundpd_epi32( __m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_mask_cvtt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_maskz_cvtt_roundpd_epi32( __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i _mm256_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm256_maskz_cvttpd_epi32( __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm_maskz_cvttpd_epi32( __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm256_cvttpd_epi32 (__m256d src);
CVTTPD2DQ __m128i _mm_cvttpd_epi32 (__m128d src);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTTPD2PI—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTTPD2PI mm, xmm/m128	RM	Valid	Valid	Convert two packer double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two packed double precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTTPD1PI __m64 _mm_cvttpd_pi32(__m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Mode Exceptions

See Table 24-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

CVTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	B	V/V	AVX512F	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2DQ (VEX.256 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96)
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])
```

VCVTTPS2DQ (VEX.128 Encoded Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] := 0
```

CVTTPS2DQ (128-bit Legacy SSE Version)

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPS2DQ __m512i _mm512_cvttps_epi32( __m512 a);
VCVTTPS2DQ __m512i _mm512_mask_cvttps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_maskz_cvttps_epi32( __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_cvtt_roundps_epi32( __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_mask_cvtt_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_maskz_cvtt_roundps_epi32( __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m256i _mm256_mask_cvttps_epi32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2DQ __m256i _mm256_maskz_cvttps_epi32( __mmask8 k, __m256 a);
VCVTTPS2DQ __m128i _mm_mask_cvttps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm_maskz_cvttps_epi32( __mmask8 k, __m128 a);
VCVTTPS2DQ __m256i _mm256_cvttps_epi32 (__m256 a)
CVTTPS2DQ __m128i _mm_cvttps_epi32 (__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTTPS2PI—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2C /r CVTTPS2PI mm, xmm/m64	RM	Valid	Valid	Convert two single precision floating-point values from xmm/m64 to two signed doubleword signed integers in mm using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two packed single precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI __m64 _mm_cvttps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 24-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

CVTTSD2SI—Convert With Truncation Scalar Double Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.LIG.F2.0F.W0 2C /r [1] VCVTTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F2.0F.W1 2C /r [1] VCVTTSD2SI r64, xmm1/m64	B	V/N.E.[2]	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	B	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSD2SI (All Versions)

```
IF 64-Bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSD2SI int _mm_cvttsd_i32( __m128d a);
VCVTTSD2SI int _mm_cvtt_roundsd_i32( __m128d a, int sae);
VCVTTSD2SI __int64 _mm_cvttsd_i64( __m128d a);
VCVTTSD2SI __int64 _mm_cvtt_roundsd_i64( __m128d a, int sae);
CVTTSD2SI int _mm_cvttsd_si32( __m128d a);
CVTTSD2SI __int64 _mm_cvttsd_si64( __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

CVTTSS2SI—Convert With Truncation Scalar Single Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.LIG.F3.0F.W0 2C /r [1] VCVTTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F3.0F.W1 2C /r [1] VCVTTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	B	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	B	V/N.E.[2]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSS2SI (All Versions)

```
IF 64-Bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSS2SI int _mm_cvttss_i32( __m128 a);
VCVTTSS2SI int _mm_cvtt_roundss_i32( __m128 a, int sae);
VCVTTSS2SI __int64 _mm_cvttss_i64( __m128 a);
VCVTTSS2SI __int64 _mm_cvtt_roundss_i64( __m128 a, int sae);
CVTTSS2SI int _mm_cvttss_si32( __m128 a);
CVTTSS2SI __int64 _mm_cvttss_si64( __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

DIV—Unsigned Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /6	DIV r/m8[1]	M	Valid	Valid	Unsigned divide AX by r/m8, with result stored in AL := Quotient, AH := Remainder.
F7 /6	DIV r/m16	M	Valid	Valid	Unsigned divide DX:AX by r/m16, with result stored in AX := Quotient, DX := Remainder.
F7 /6	DIV r/m32	M	Valid	Valid	Unsigned divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder.
REX.W + F7 /6	DIV r/m64	M	Valid	N.E.	Unsigned divide RDX:RAX by r/m64, with result stored in RAX := Quotient, RDX := Remainder.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-24.

Table 3-24. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} – 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	$2^{64} – 1$

Operation

```
IF SRC = 0
    THEN #DE; FI; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
        temp := AX / SRC;
        IF temp > FFH
            THEN #DE; (* Divide error *)
            ELSE
                AL := temp;
                AH := AX MOD SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp := DX:AX / SRC;
            IF temp > FFFFH
                THEN #DE; (* Divide error *)
            ELSE
                AX := temp;
                DX := DX:AX MOD SRC;
            FI;
        FI;
    ELSE IF Operandsize = 32 (* Quadword/doubleword operation *)
        THEN
            temp := EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                EAX := temp;
                EDX := EDX:EAX MOD SRC;
            FI;
        FI;
    ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp := RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                RAX := temp;
                RDX := RDX:RAX MOD SRC;
            FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

DIVPD—Divide Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	A	V/V	SSE2	Divide packed double precision floating-point values in xmm1 by packed double precision floating-point values in xmm2/mem.
VEX.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/mem.
VEX.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/mem.
EVEX.128.66.0F.W1 5E /r VDIVPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5E /r VDIVPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1	Divide packed double precision floating-point values in zmm2 by packed double precision floating-point values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD divide of the double precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand (the second operand) is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.128 encoded version: The first source operand (the second operand) is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination are zeroed.

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL-1:128) of the corresponding destination are unmodified.

Operation

VDIVPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures
Software Developer’s Manual, Volume 1
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
       THEN
           IF (EVEX.b = 1) AND (SRC2 *is memory*)
               THEN
                   DEST[i+63:i] := SRC1[i+63:i] / SRC2[63:0]
               ELSE
                   DEST[i+63:i] := SRC1[i+63:i] / SRC2[i+63:i]
           FI;
       ELSE
           IF *merging-masking*                    ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
               ELSE                                ; zeroing-masking
                   DEST[i+63:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VDIVPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[191:128] := SRC1[191:128] / SRC2[191:128]
DEST[255:192] := SRC1[255:192] / SRC2[255:192]
DEST[MAXVL-1:256] := 0;
```

VDIVPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[MAXVL-1:128] := 0;
```

DIVPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64] / SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPD __m512d _mm512_div_pd(__m512d a, __m512d b);
VDIVPD __m512d _mm512_mask_div_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d _mm512_maskz_div_pd(__mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d _mm512_div_round_pd(__m512d a, __m512d b, int);
VDIVPD __m512d _mm512_mask_div_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_maskz_div_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d _mm256_div_pd (__m256d a, __m256d b);
DIVPD __m128d _mm_div_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

DIVPS—Divide Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5E /r DIVPS xmm1, xmm2/m128	A	V/V	SSE	Divide packed single precision floating-point values in xmm1 by packed single precision floating-point values in xmm2/mem.
VEX.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/mem.
VEX.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/mem.
EVEX.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F OR AVX10.1	Divide packed single precision floating-point values in zmm2 by packed single precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD divide of the four, eight or sixteen packed single precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VDIVPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
               DEST[i+31:i] := SRC1[i+31:i] / SRC2[31:0]
            ELSE
               DEST[i+31:i] := SRC1[i+31:i] / SRC2[i+31:i]
         FI;
      ELSE
         IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
               DEST[i+31:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VDIVPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[159:128] := SRC1[159:128] / SRC2[159:128]
DEST[191:160] := SRC1[191:160] / SRC2[191:160]
DEST[223:192] := SRC1[223:192] / SRC2[223:192]
DEST[255:224] := SRC1[255:224] / SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VDIVPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

DIVPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[63:32] := SRC1[63:32] / SRC2[63:32]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[127:96] := SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVPS __m512 _mm512_div_ps( __m512 a, __m512 b);
VDIVPS __m512 _mm512_mask_div_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VDIVPS __m512 _mm512_maskz_div_ps(__mmask16 k, __m512 a, __m512 b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPS __m512 _mm512_div_round_ps( __m512 a, __m512 b, int);
VDIVPS __m512 _mm512_mask_div_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m512 _mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m256 _mm256_div_ps (__m256 a, __m256 b);
DIVPS __m128 _mm_div_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

DIVSD—Divide Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	A	V/V	SSE2	Divide low double precision floating-point value in xmm1 by low double precision floating-point value in xmm2/m64.
VEX.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.
EVEX.LLIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Divides the low double precision floating-point value in the first source operand by the low double precision floating-point value in the second source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN      DEST[63:0] := SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VDIVSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

DIVSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd( __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd( __m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd( __mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

DIVSS—Divide Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	A	V/V	SSE	Divide low single precision floating-point value in xmm1 by low single precision floating-point value in xmm2/m32.
VEX.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.
EVEX.LLIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Divides the low single precision floating-point value in the first source operand by the low single precision floating-point value in the second source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VDIVSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

DIVSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] / SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss( __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss( __m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss( __mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

EXTRACTPS—Extract Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8	A	VV	SSE4_1	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	A	V/V	AVX	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
EVEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	B	V/V	AVX512F OR AVX10.1	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Extracts a single precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

VEX.vvvv/EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

VEXTRACTPS (EVEX and VEX.128 Encoded Version)

```
SRC_OFFSET := IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] := 0
ELSE
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

EXTRACTPS (128-bit Legacy SSE Version)

```
SRC_OFFSET := IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] := 0
ELSE
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS int _mm_extract_ps (__m128 a, const int nidx);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-59, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

GF2P8AFFINEINVQB—Galois Field Affine Transformation Inverse

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	(AVX512F OR AVX10.1) GFNI	Computes inverse affine transformation in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEINVB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * inv(x) + b where "A" is an 8 by 8 bit matrix, and "x" and "b" are 8-bit vectors. The inverse of the bytes in x is defined with respect to the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

One SIMD register (operand 1) holds "x" as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 "A" values, which are operated upon by the correspondingly aligned 8 "x" values in the first register. The "b" vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

The inverse of each byte is given by the following table. The upper nibble is on the vertical axis and the lower nibble is on the horizontal axis. For example, the inverse of 0x95 is 0x8A.

Table 3-59. Inverse Byte Listings

-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	9
5	ED	5C	5	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	6	A1	FA	81	82
8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	2	B9	A4
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
B	C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
C	B	28	2F	A3	DA	D4	E4	F	A9	27	53	4	1B	FC	AC	E6
D	7A	7	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
E	B1	D	D6	EB	C6	E	CF	AD	8	4E	D7	E3	5D	50	1E	B3
F	5B	23	38	34	68	46	3	8C	DD	9C	7D	A0	CD	1A	41	1C

Operation

```
define affine_inverse_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        * inverse(x) is defined in the table above *
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND inverse(src1byte)) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 := SRC2.qword[0]
    ELSE:
        tsrc2 := SRC2.qword[j]

FOR b := 0 to 7:
    IF k1[j*8+b] OR *no writemask*:
        FOR i := 0 to 7:
            DEST.qword[j].byte[b] := affine_inverse_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
    ELSE IF *zeroing*:
        DEST.qword[j].byte[b] := 0
    *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_inverse_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] := 0
```

GF2P8AFFINEINVQB srcdest, src1, imm8 (128b SSE Encoded Version)

```
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRCDEST.qword[j].byte[b] := affine_inverse_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8AFFINEINVQB __m128i _mm_gf2p8affineinv_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_mask_gf2p8affineinv_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_maskz_gf2p8affineinv_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGF2P8AFFINEINVQB __m256i _mm256_gf2p8affineinv_epi64_epi8(__m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_mask_gf2p8affineinv_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_maskz_gf2p8affineinv_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m512i _mm512_gf2p8affineinv_epi64_epi8(__m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_mask_gf2p8affineinv_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_maskz_gf2p8affineinv_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-52, "Type E4NF Class Exception Conditions."

GF2P8AFFINEQB—Galois Field Affine Transformation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CE /r /ib GF2P8AFFINEQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL OR AVX10.1) GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.512.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	(AVX512F OR AVX10.1) GFNI	Computes affine transformation in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * x + b where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define parity(x):
    t := 0                  // single bit
    FOR i := 0 to 7:
        t = t xor x.bit[i]
    return t

define affine_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND src1byte) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 := SRC2.qword[0]
    ELSE:
        tsrc2 := SRC2.qword[j]

    FOR b := 0 to 7:
        IF k1[j*8+b] OR *no writemask*:
            DEST.qword[j].byte[b] := affine_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
        ELSE IF *zeroing*:
            DEST.qword[j].byte[b] := 0
        *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] := 0
```

GF2P8AFFINEQB srcdest, src1, imm8 (128b SSE Encoded Version)

```
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRCDEST.qword[j].byte[b] := affine_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)GF2P8AFFINEQB __m128i _mm_gf2p8affine_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_mask_gf2p8affine_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_maskz_gf2p8affine_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGF2P8AFFINEQB __m256i _mm256_gf2p8affine_epi64_epi8(__m256i, __m256i, int);
VGF2P8AFFINEQB __m256i _mm256_mask_gf2p8affine_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
VGF2P8AFFINEQB __m256i _mm256_maskz_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGF2P8AFFINEQB __m512i _mm512_gf2p8affine_epi64_epi8(__m512i, __m512i, int);
VGF2P8AFFINEQB __m512i _mm512_mask_gf2p8affine_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
VGF2P8AFFINEQB __m512i _mm512_maskz_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-52, “Type E4NF Class Exception Conditions.”

GF2P8MULB—Galois Field Multiply Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F38 CF /r GF2P8MULB xmm1, xmm2/m128	A	V/V	GFNI	Multiplies elements in the finite field GF(2^8).
VEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1, xmm2, xmm3/m128	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
VEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1, ymm2, ymm3/m256	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL OR AVX10.1) GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL OR AVX10.1) GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.512.66.0F38.W0 CF /r VGF2P8MULB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	(AVX512F OR AVX10.1) GFNI	Multiplies elements in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The instruction multiplies elements in the finite field GF(2^8), operating on a byte (field element) in the first source operand and the corresponding byte in a second source operand. The field GF(2^8) is represented in polynomial representation with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

This instruction does not support broadcasting.

The EVEX encoded form of this instruction supports memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define gf2p8mul_byte(src1byte, src2byte):
    tword := 0
    FOR i := 0 to 7:
        IF src2byte.bit[i]:
            tword := tword XOR (src1byte<< i)
        * carry out polynomial reduction by the characteristic polynomial p*
    FOR i := 14 downto 8:
        p := 0x11B << (i-8)            *0x11B = 0000_0001_0001_1011 in binary*
        IF tword.bit[i]:
            tword := tword XOR p
return tword.byte[0]
```

VGF2P8MULB dest, src1, src2 (EVEX Encoded Version)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
    ELSE IF *zeroing*:
        DEST.byte[j] := 0
    * ELSE DEST.byte[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VGF2P8MULB dest, src1, src2 (128b and 256b VEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256)
FOR j := 0 TO KL-1:
    DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
DEST[MAX_VL-1:VL] := 0
```

GF2P8MULB srcdest, src1 (128b SSE Encoded Version)

```
FOR j := 0 TO 15:
    SRCDEST.byte[j] :=gf2p8mul_byte(SRCDEST.byte[j], SRC1.byte[j])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)GF2P8MULB __m128i _mm_gf2p8mul_epi8(__m128i, __m128i);
(V)GF2P8MULB __m128i _mm_mask_gf2p8mul_epi8(__m128i, __mmask16, __m128i, __m128i);
(V)GF2P8MULB __m128i _mm_maskz_gf2p8mul_epi8(__mmask16, __m128i, __m128i);
VGF2P8MULB __m256i _mm256_gf2p8mul_epi8(__m256i, __m256i);
VGF2P8MULB __m256i _mm256_mask_gf2p8mul_epi8(__m256i, __mmask32, __m256i, __m256i);
VGF2P8MULB __m256i _mm256_maskz_gf2p8mul_epi8(__mmask32, __m256i, __m256i);
VGF2P8MULB __m512i _mm512_gf2p8mul_epi8(__m512i, __m512i);
VGF2P8MULB __m512i _mm512_mask_gf2p8mul_epi8(__m512i, __mmask64, __m512i, __m512i);
VGF2P8MULB __m512i _mm512_maskz_gf2p8mul_epi8(__mmask64, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-51, “Type E4 Class Exception Conditions.”

IDIV—Signed Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /7	IDIV r/m8[1]	M	Valid	Valid	Signed divide AX by r/m8, with result stored in: AL := Quotient, AH := Remainder.
F7 /7	IDIV r/m16	M	Valid	Valid	Signed divide DX:AX by r/m16, with result stored in AX := Quotient, DX := Remainder.
F7 /7	IDIV r/m32	M	Valid	Valid	Signed divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder.
REX.W + F7 /7	IDIV r/m64	M	Valid	N.E.	Signed divide RDX:RAX by r/m64, with result stored in RAX := Quotient, RDX := Remainder.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-60.

Table 3-60. IDIV Results

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	−128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	−32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{31} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	-2^{63} to $2^{63} - 1$

Operation

```
IF SRC = 0
    THEN #DE; (* Divide error *)
FI;
IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp := AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
        (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL := temp;
                AH := AX SignedModulus SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp := DX:AX / SRC; (* Signed division *)
            IF (temp > 7FFFH) or (temp < 8000H)
            (* If a positive result is greater than 7FFFH
            or a negative result is less than 8000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    AX := temp;
                    DX := DX:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
            temp := EDX:EAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFH) or (temp < 80000000H)
            (* If a positive result is greater than 7FFFFFFFH
            or a negative result is less than 80000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    EAX := temp;
                    EDX := EDXE:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
            temp := RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
            (* If a positive result is greater than 7FFFFFFFFFFFFFFFH
            or a negative result is less than 8000000000000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    RAX := temp;
                    RDX := RDE:RAX SignedModulus SRC;
            FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL r/m8[1]	M	Valid	Valid	AX:= AL ∗ r/m byte.
F7 /5	IMUL r/m16	M	Valid	Valid	DX:AX := AX ∗ r/m word.
F7 /5	IMUL r/m32	M	Valid	Valid	EDX:EAX := EAX ∗ r/m32.
REX.W + F7 /5	IMUL r/m64	M	Valid	N.E.	RDX:RAX := RAX ∗ r/m64.
0F AF /r	IMUL r16, r/m16	RM	Valid	Valid	Word register := word register ∗ r/m16.
0F AF /r	IMUL r32, r/m32	RM	Valid	Valid	Doubleword register := doubleword register ∗ r/m32.
REX.W + 0F AF /r	IMUL r64, r/m64	RM	Valid	N.E.	Quadword register := Quadword register ∗ r/m64.
6B /r ib	IMUL r16, r/m16, imm8	RMI	Valid	Valid	Word register := r/m16 ∗ sign-extended immediate byte.
6B /r ib	IMUL r32, r/m32, imm8	RMI	Valid	Valid	Doubleword register := r/m32 ∗ sign-extended immediate byte.
REX.W + 6B /r ib	IMUL r64, r/m64, imm8	RMI	Valid	N.E.	Quadword register := r/m64 ∗ sign-extended immediate byte.
69 /r iw	IMUL r16, r/m16, imm16	RMI	Valid	Valid	Word register := r/m16 ∗ immediate word.
69 /r id	IMUL r32, r/m32, imm32	RMI	Valid	Valid	Doubleword register := r/m32 ∗ immediate doubleword.
REX.W + 69 /r id	IMUL r64, r/m64, imm32	RMI	Valid	N.E.	Quadword register := r/m64 ∗ immediate doubleword.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	N/A

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

Operation

```
IF (NumberOfOperands = 1)
    THEN IF (OperandSize = 8)
        THEN
            TMP_XP := AL ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
            AX := TMP_XP[15:0];
            IF SignExtend(TMP_XP[7:0]) = TMP_XP
                THEN CF := 0; OF := 0;
                ELSE CF := 1; OF := 1; FI;
        ELSE IF OperandSize = 16
            THEN
                TMP_XP := AX ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
                DX:AX := TMP_XP[31:0];
                IF SignExtend(TMP_XP[15:0]) = TMP_XP
                    THEN CF := 0; OF := 0;
                    ELSE CF := 1; OF := 1; FI;
            ELSE IF OperandSize = 32
                THEN
                    TMP_XP := EAX ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
                    EDX:EAX := TMP_XP[63:0];
                    IF SignExtend(TMP_XP[31:0]) = TMP_XP
                        THEN CF := 0; OF := 0;
                        ELSE CF := 1; OF := 1; FI;
                ELSE (* OperandSize = 64 *)
                    TMP_XP := RAX ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
                    EDX:EAX := TMP_XP[127:0];
                    IF SignExtend(TMP_XP[63:0]) = TMP_XP
                        THEN CF := 0; OF := 0;
                        ELSE CF := 1; OF := 1; FI;
                FI;
        FI;
    ELSE IF (NumberOfOperands = 2)
        THEN
            TMP_XP := DEST ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
            DEST := TruncateToOperandSize(TMP_XP);
            IF SignExtend(DEST) ≠ TMP_XP
                THEN CF := 1; OF := 1;
                ELSE CF := 0; OF := 0; FI;
        ELSE (* NumberOfOperands = 3 *)
            TMP_XP := SRC1 ∗ SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
            DEST := TruncateToOperandSize(TMP_XP);
            IF SignExtend(DEST) ≠ TMP_XP
                THEN CF := 1; OF := 1;
                ELSE CF := 0; OF := 0; FI;
    FI;
FI;
```

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC r/m8[1]	M	Valid	Valid	Increment r/m byte by 1.
FF /0	INC r/m16	M	Valid	Valid	Increment r/m word by 1.
FF /0	INC r/m32	M	Valid	Valid	Increment r/m doubleword by 1.
REX.W + FF /0	INC r/m64	M	Valid	N.E.	Increment r/m quadword by 1.
40+ rw[2]	INC r16	O	N.E.	Valid	Increment word register by 1.
40+ rd	INC r32	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A
O	opcode + rd (r, w)	N/A	N/A	N/A

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC r16 and INC r32 are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

Operation

```
DEST := DEST + 1;
```

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULLsegment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

INSERTPS—Insert Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8	A	V/V	SSE4_1	Insert a single precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.
VEX.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.
EVEX.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	C	V/V	AVX512F OR AVX10.1	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(register source form)

Copy a single precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count_S bits are ignored.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

VINSERTPS (VEX.128 and EVEX Encoded Version)

```
IF (SRC = REG) THEN COUNT_S := imm8[7:6]
    ELSE COUNT_S := 0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
    0: TMP := SRC2[31:0]
    1: TMP := SRC2[63:32]
    2: TMP := SRC2[95:64]
    3: TMP := SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
    0: TMP2[31:0] := TMP
        TMP2[127:32] := SRC1[127:32]
    1: TMP2[63:32] := TMP
        TMP2[31:0] := SRC1[31:0]
        TMP2[127:64] := SRC1[127:64]
    2: TMP2[95:64] := TMP
        TMP2[63:0] := SRC1[63:0]
        TMP2[127:96] := SRC1[127:96]
    3: TMP2[127:96] := TMP
        TMP2[95:0] := SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
    ELSE DEST[31:0] := TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] := 00000000H
    ELSE DEST[63:32] := TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] := 00000000H
    ELSE DEST[95:64] := TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] := 00000000H
    ELSE DEST[127:96] := TMP2[127:96]
DEST[MAXVL-1:128] := 0
```

INSERTPS (128-bit Legacy SSE Version)

```
IF (SRC = REG) THEN COUNT_S :=imm8[7:6]
    ELSE COUNT_S :=0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
    0: TMP := SRC[31:0]
    1: TMP := SRC[63:32]
    2: TMP := SRC[95:64]
    3: TMP := SRC[127:96]
ESAC;

CASE (COUNT_D) OF
    0: TMP2[31:0] := TMP
        TMP2[127:32] := DEST[127:32]
    1: TMP2[63:32] := TMP
        TMP2[31:0] := DEST[31:0]
        TMP2[127:64] := DEST[127:64]
    2: TMP2[95:64] := TMP
```

```
            TMP2[63:0] := DEST[63:0]
            TMP2[127:96] := DEST[127:96]
     3: TMP2[127:96] := TMP
            TMP2[95:0] := DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
     ELSE DEST[31:0] := TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] := 00000000H
     ELSE DEST[63:32] := TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] := 00000000H
     ELSE DEST[95:64] := TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] := 00000000H
     ELSE DEST[127:96] := TMP2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VINSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

LAR—Load Access Rights

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 02 /r	LAR r16, r16/m16	RM	Valid	Valid	Load access rights from specified descriptor.
0F 02 /r	LAR r32, r32/m16[1]	RM	Valid	Valid	Load access rights from specified descriptor.
REX.W + 0F 02/r	LAR r32, r64/m16[1]	RM	Valid	N.E.	Load access rights from specified descriptor.

NOTES:

1. Regardless of operand size, only bits 15:0 of a register source operand are used. Other bits are ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
  - — Bits 19:16 are undefined.
  - — Bit 20 returns the software-available bit in the descriptor.
  - — Bit 21 returns the L flag.
  - — Bit 22 returns the D/B flag.
  - — Bit 23 returns the G flag.
  - — Bits 31:24 are returned as 0.

When the operand size is 16 bits, only the low 16 bits identified above are returned; the upper bits of the destination are unmodified. When the operand size is 32 bits, the 32-bit value identified above is loaded into the destination operand; the upper bits of the destination are cleared. When the operand is 64 bits, the 32-bit value is zero-extended to 64 bits and loaded into the destination operand. (The behavior with 32-bit and 64-bit operand sizes is identical.)

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed

- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-62.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode and IA-32e mode.

Table 3-62. Segment and Gate Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	Yes	Reserved	No
5	16-bit/32-bit task gate	Yes	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	Available 64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	Yes	64-bit call gate	Yes
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

```
IF Offset(SRC) > descriptor table limit
    THEN
        ZF := 0;
    ELSE
        SegmentDescriptor := descriptor referenced by SRC;
        IF SegmentDescriptor(Type) ≠ conforming code segment
        and (CPL > DPL) or (RPL > DPL)
        or SegmentDescriptor(Type) is not valid for instruction
            THEN
                ZF := 0;
            ELSE
                DEST := access rights from SegmentDescriptor as given in Description section;
                ZF := 1;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LAR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C5 /r	LDS r16,m16:16	RM	Invalid	Valid	Load DS:r16 with far pointer from memory.
C5 /r	LDS r32,m16:32	RM	Invalid	Valid	Load DS:r32 with far pointer from memory.
0F B2 /r	LSS r16,m16:16	RM	Valid	Valid	Load SS:r16 with far pointer from memory.
0F B2 /r	LSS r32,m16:32	RM	Valid	Valid	Load SS:r32 with far pointer from memory.
REX.W + 0F B2 /r	LSS r64,m16:64	RM	Valid	N.E.	Load SS:r64 with far pointer from memory.
C4 /r	LES r16,m16:16	RM	Invalid	Valid	Load ES:r16 with far pointer from memory.
C4 /r	LES r32,m16:32	RM	Invalid	Valid	Load ES:r32 with far pointer from memory.
0F B4 /r	LFS r16,m16:16	RM	Valid	Valid	Load FS:r16 with far pointer from memory.
0F B4 /r	LFS r32,m16:32	RM	Valid	Valid	Load FS:r32 with far pointer from memory.
REX.W + 0F B4 /r	LFS r64,m16:64	RM	Valid	N.E.	Load FS:r64 with far pointer from memory.
0F B5 /r	LGS r16,m16:16	RM	Valid	Valid	Load GS:r16 with far pointer from memory.
0F B5 /r	LGS r32,m16:32	RM	Valid	Valid	Load GS:r32 with far pointer from memory.
REX.W + 0F B5 /r	LGS r64,m16:64	RM	Valid	N.E.	Load GS:r64 with far pointer from memory.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
64-BIT_MODE
   IF SS is loaded
      THEN
         IF SegmentSelector = NULL and ( (RPL = 3) or
                  (RPL ≠ 3 and RPL ≠ CPL) )
            THEN #GP(0);
         ELSE IF descriptor is in non-canonical space
```

```
                THEN #GP(selector); FI;
            ELSE IF Segment selector index is not within descriptor table limits
                    or segment selector RPL ≠ CPL
                    or access rights indicate nonwritable data segment
                    or DPL ≠ CPL
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
                THEN #SS(selector); FI;
            FI;
            SS := SegmentSelector(SRC);
            SS := SegmentDescriptor([SRC]);
    ELSE IF attempt to load DS, or ES
        THEN #UD;
    ELSE IF FS, or GS is loaded with non-NULL segment selector
        THEN IF Segment selector index is not within descriptor table limits
            or access rights indicate segment neither data nor readable code segment
            or segment is data or nonconforming-code segment
            and ( RPL > DPL or CPL > DPL)
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
                THEN #NP(selector); FI;
            FI;
            SegmentRegister := SegmentSelector(SRC) ;
            SegmentRegister := SegmentDescriptor([SRC]);
        FI;
    ELSE IF FS, or GS is loaded with a NULL selector:
        THEN
            SegmentRegister := NULLSelector;
            SegmentRegister(DescriptorValidBit) := 0; FI; (* Hidden flag;
                not accessible by software *)
    FI;
    DEST := Offset(SRC);
PREOTECTED MODE OR COMPATIBILITY MODE;
    IF SS is loaded
        THEN
            IF SegementSelector = NULL
                THEN #GP(0);
            ELSE IF Segment selector index is not within descriptor table limits
                    or segment selector RPL ≠ CPL
                    or access rights indicate nonwritable data segment
                    or DPL ≠ CPL
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
                THEN #SS(selector); FI;
            FI;
            SS := SegmentSelector(SRC);
            SS := SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
        THEN IF Segment selector index is not within descriptor table limits
            or access rights indicate segment neither data nor readable code segment
            or segment is data or nonconforming-code segment
            and (RPL > DPL or CPL > DPL)
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
```

```
                THEN #NP(selector); FI;
            FI;
            SegmentRegister := SegmentSelector(SRC) AND RPL;
            SegmentRegister := SegmentDescriptor([SRC]);
        FI;
    ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
        THEN
            SegmentRegister := NULLSelector;
            SegmentRegister(DescriptorValidBit) := 0; FI; (* Hidden flag;
                not accessible by software *)
    FI;
    DEST := Offset(SRC);
Real-Address or Virtual-8086 Mode
    SegmentRegister := SegmentSelector(SRC); FI;
    DEST := Offset(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a NULL selector is loaded into the SS register.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL.
	If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a NULL selector is attempted to be loaded into the SS register in compatibility mode.
	If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode.
	If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.
#GP(Selector)	If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.
#SS(0)	If a memory operand effective address is non-canonical
#SS(Selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

LSL—Load Segment Limit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 03 /r	LSL r16, r16/m16	RM	Valid	Valid	Load segment limit from specified descriptor.
0F 03 /r	LSL r32, r32/m16[1]	RM	Valid	Valid	Load segment limit from specified descriptor.
REX.W + 0F 03 /r	LSL r64, r32/m16[1]	RM	Valid	Valid	Load segment limit from specified descriptor.

NOTES:

1. Regardless of operand size, only bits 15:0 of a register operand are used. Other bits are ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Loads the segment limit from the segment descriptor (see below) specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit) as read from the descriptor. If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit “raw” limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 16 bits, a valid 32-bit byte limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand; the upper bits of the destination are unmodified. When the operand size is 32 bits, the 32-bit byte limit is loaded into the destination operand; the upper bits of the destination are cleared. When the operand is 64 bits, the 32-bit byte limit is zero-extended to 64 bits and loaded into the destination operand. (The behavior with 32-bit and 64-bit operand sizes is identical.)

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the Table 3-66.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

Table 3-66. Segment and Gate Descriptor Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT[1]	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	No	Reserved	No
5	16-bit/32-bit task gate	No	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	64-bit TSS[1]	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS[1]	Yes
C	32-bit call gate	No	64-bit call gate	No
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

NOTES:

1. In this case, the descriptor comprises 16 bytes; bits 12:8 of the upper 4 bytes must be 0.

Operation

```
IF SRC(Offset) > descriptor table limit
    THEN ZF := 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
and (CPL > DPL) OR (RPL > DPL)
or Segment type is not valid for instruction
        THEN
            ZF := 0;
        ELSE
            temp := SegmentLimit([SRC]);
            IF (SegmentDescriptor(G) = 1)
                THEN temp := (temp << 12) OR 00000FFFH;
            ELSE IF OperandSize = 32
                THEN DEST := temp; FI;
            ELSE IF OperandSize = 64 (* REX.W used *)
                THEN DEST := temp(* Zero-extended *); FI;
            ELSE (* OperandSize = 16 *)
                DEST := temp AND FFFFH;
            FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0. The CF, OF, SF, AF, and PF flags are not modified.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LSL instruction cannot be executed in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LSL instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LZCNT—Count the Number of Leading Zero Bits

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F BD /r LZCNT r16, r/m16	RM	V/V	LZCNT	Count the number of leading zero bits in r/m16, return result in r16.
F3 0F BD /r LZCNT r32, r/m32	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
F3 REX.W 0F BD /r LZCNT r64, r/m64	RM	V/N.E.	LZCNT	Count the number of leading zero bits in r/m64, return result in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

LZCNT counts the number of leading most significant zero bits in a source operand (second operand) and returns the result in the destination (first operand). LZCNT is an extension of the BSR instruction. The key difference between the LZCNT and BSR instructions is that when the source operand is zero, LZCNT outputs the operand size to the destination operand, whereas BSR leaves the destination operand unmodified.

On processors that do not support LZCNT, the instruction byte encoding is executed as BSR.

Operation

```
temp := OperandSize - 1
DEST := 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
DO
    temp := temp - 1
    DEST := DEST+ 1
OD

IF DEST = OperandSize
    CF := 1
ELSE
    CF := 0
FI

IF DEST = 0
    ZF := 1
ELSE
    ZF := 0
FI
```

Flags Affected

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF, and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```
LZCNT unsigned __int32 _lzcnt_u32(unsigned __int32 src);
LZCNT unsigned __int64 _lzcnt_u64(unsigned __int64 src);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#UD	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

7. Updates to Chapter 4, Volume 2B

Change bars and violet text show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, M-U.

Changes to this chapter:

- Revised opcode tables removing REX+ prefixes for instructions: MOV, MOVSX/MOVSXD, MOVZX, MUL, NEG, NOT, OR, RCL/RCR/ROL/ROR, SAL/SAR/SHL/SHR, SBB, SETcc, SUB, TEST.
- Updated opcode table and revised description for PCLMULQDQ for YMM and ZMM.
- Split the instruction REP/REPE/REPZ/REPNE/REPNZ-Repeat String Operation Prefix into REP-Repeat String Operation prefix instruction, REPE/REPZ-Repeat String Operation While Zero prefix instruction, and REPNE/REPNZ-Repeat String Operation While Not Zero prefix instruction.
- Revised Description of TZCNT instruction.
- Removed footnote references to verify vector options for the following instructions:
  — MAXPD
  — MAXPS
  — MAXSD
  — MAXSS
  — MINPD
  — MINPS
  — MINSD
  — MINSS
  — MOVAPD
  — MOVAPS
  — MOVDDUP
  — MOVD/MOVQ
  — MOVDQA, VMOVDQA32/64
  — MOVDQU, VMOVDQU8/16/32/64
  — MOVHLPS
  — MOVHPD
  — MOVHPS
  — MOVLHPS
  — MOVLPD
  — MOVLPS
  — MOVNTDQ
  — MOVNTDQA
  — MOVNTPD
  — MOVNTPS
  — MOVQ
  — MOVSD
  — MOVSHDUP
  — MOVSLDUP
  — MOVSS
  — MOVUPD
  — MOVUPS
  — MULPD
  — MULPS

— MULSD
— MULSS
— ORPD
— ORPS
— PABSB/PABSW/PABSD/PABSQ
— PACKSSWB/PACKSSDW
— PACKUSDW
— PACKUSWB
— PADDB/PADDW/PADDD/PADDQ
— PADDSB/PADDSW
— PADDUSB/PADDUSW
— PALIGNR
— PAND
— PANDN
— PAVGB/PAVGW
— PCLMULQDQ
— PCMPEQB/PCMPEQW/PCMPEQD
— PCMPEQQ
— PCMPGTB/PCMPGTW/PCMPGTD
— PCMPGTQ
— PEXTRB/PEXTRD/PEXTRQ
— PEXTRW
— PINSRB/PINSRD/PINSRQ
— PINSRW
— PMADDUBSW
— PMADDWD
— PMAXSB/PMAXSW/PMAXSD/PMAXSQ
— PMAXUB/PMAXUW
— PMAXUD/PMAXUQ
— PMINSB/PMINSW
— PMINSD/PMINSQ
— PMINUB/PMINUW
— PMINUD/PMINUQ
— PMOVSX
— PMOVZX
— PMULDQ
— PMULHRSW
— PMULHUW
— PMULHW
— PMULLD/PMULLQ
— PMULLW
— PMULUDQ
— POR
— PSADBW
— PSHUFB
— PSHUFD

— PSHUFHW
— PSHUFLW
— PSLLDQ
— PSLLW/PSLLD/PSLLQ
— PSRAW/PSRAD/PSRAQ
— PSRLDQ
— PSRLW/PSRLD/PSRLQ
— PSUBB/PSUBW/PSUBD
— PSUBQ
— PSUBSB/PSUBSW
— PSUBUSB/PSUBUSW
— PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ
— PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ
— PXOR
— SHUFPD
— SHUFPS
— SQRTPD
— SQRTPS
— SQRTSD
— SQRTSS
— SUBPD
— SUBPS
— SUBSD
— SUBSS
— UCOMISD
— UCOMISS
— UNPCKHPD
— UNPCKHPS
— UNPCKLPD
— UNPCKLPS

CHAPTER 4
INSTRUCTION SET REFERENCE, M-U

4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM

The notations introduced in this section are referenced in the reference pages of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM. The operation of the immediate control byte is common to these four string text processing instructions of SSE4.2. This section describes the common operations.

4.1.1 General Description

The operation of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM is defined by the combination of the respective opcode and the interpretation of an immediate control byte that is part of the instruction encoding.

The opcode controls the relationship of input bytes/words to each other (determines whether the inputs terminated strings or whether lengths are expressed explicitly) as well as the desired output (index or mask).

The imm8 control byte for PCMPESTRM/PCMPESTRI/PCMPISTRM/PCMPISTRI encodes a significant amount of programmable control over the functionality of those instructions. Some functionality is unique to each instruction while some is common across some or all of the four instructions. This section describes functionality which is common across the four instructions.

The arithmetic flags (ZF, CF, SF, OF, AF, PF) are set as a result of these instructions. However, the meanings of the flags have been overloaded from their typical meanings in order to provide additional information regarding the relationships of the two inputs.

PCMPxSTRx instructions perform arithmetic comparisons between all possible pairs of bytes or words, one from each packed input source operand. The boolean results of those comparisons are then aggregated in order to produce meaningful results. The imm8 control byte is used to affect the interpretation of individual input elements as well as control the arithmetic comparisons used and the specific aggregation scheme.

Specifically, the imm8 Control Byte consists of bit fields that control the following attributes:

- **Source data format —** Byte/word data element granularity, signed or unsigned elements.
- **Aggregation operation —** Encodes the mode of per-element comparison operation and the aggregation of per-element comparisons into an intermediate result.
- **Polarity —** Specifies intermediate processing to be performed on the intermediate result.
- **Output selection —** Specifies final operation to produce the output (depending on index or mask) from the intermediate result.

4.1.2 Source Data Format

Table 4-1. Source Data Format

Imm8[1:0]	Meaning	Description
00b	Unsigned bytes	Both 128-bit sources are treated as packed, unsigned bytes.
01b	Unsigned words	Both 128-bit sources are treated as packed, unsigned words.
10b	Signed bytes	Both 128-bit sources are treated as packed, signed bytes.
11b	Signed words	Both 128-bit sources are treated as packed, signed words.

If the imm8 control byte has bit[0] cleared, each source contains 16 packed bytes. If the bit is set each source contains 8 packed words. If the imm8 control byte has bit[1] cleared, each input contains unsigned data. If the bit is set each source contains signed data.

4.1.3 Aggregation Operation

Table 4-2. Aggregation Operation

Imm8[3:2]	Mode	Comparison
00b	Equal any	The arithmetic comparison is “equal.”
01b	Ranges	Arithmetic comparison is “greater than or equal” between even indexed bytes/words of reg and each byte/word of reg/mem. Arithmetic comparison is “less than or equal” between odd indexed bytes/words of reg and each byte/word of reg/mem. (reg/mem[m] >= reg[n] for n = even, reg/mem[m] <= reg[n] for n = odd)
10b	Equal each	The arithmetic comparison is “equal.”
11b	Equal ordered	The arithmetic comparison is “equal.”

All 256 (64) possible comparisons are always performed. The individual Boolean results of those comparisons are referred by “BoolRes[*Reg/Mem element index, Reg element index*].” Comparisons evaluating to “True” are represented with a 1, False with a 0 (positive logic). The initial results are then aggregated into a 16-bit (8-bit) intermediate result (IntRes1) using one of the modes described in the table below, as determined by imm8 control byte bits[3:2].

See Section 4.1.6 for a description of the overrideIfDataInvalid() function used in Table 4-3.

Table 4-3. Aggregation Operation

Mode	Pseudocode

Equal any
(find characters from a set)
	UpperBound = imm8[0] ? 7 : 15;
	IntRes1 = 0;
	For j = 0 to UpperBound, j++
	  For i = 0 to UpperBound, i++
	    IntRes1[j] OR= overrideIfDataInvalid(BoolRes[j,i])

Ranges
(find characters from ranges)
	UpperBound = imm8[0] ? 7 : 15;
	IntRes1 = 0;
	For j = 0 to UpperBound, j++
	  For i = 0 to UpperBound, i+=2
	    IntRes1[j] OR= (overrideIfDataInvalid(BoolRes[j,i]) AND overrideIfDataInvalid(BoolRes[j,i+1]))

Table 4-3. Aggregation Operation (Contd.)

Equal each
(string compare)
	UpperBound = imm8[0] ? 7 : 15;
	IntRes1 = 0;
	For i = 0 to UpperBound, i++
	   IntRes1[i] = overrideIfDataInvalid(BoolRes[i,i])

Equal ordered
(substring search)
	UpperBound = imm8[0] ? 7 :15;
	IntRes1 = imm8[0] ? FFH : FFFFH
	For j = 0 to UpperBound, j++
	   For i = 0 to UpperBound-j, k=j to UpperBound, k++, i++
	      IntRes1[j] AND= overrideIfDataInvalid(BoolRes[k,i])

4.1.4 Polarity

IntRes1 may then be further modified by performing a 1's complement, according to the value of the imm8 control byte bit[4]. Optionally, a mask may be used such that only those IntRes1 bits which correspond to "valid" reg/mem input elements are complemented (note that the definition of a valid input element is dependent on the specific opcode and is defined in each opcode's description). The result of the possible negation is referred to as IntRes2.

Table 4-4. Polarity

Imm8[5:4]	Operation	Description
00b	Positive Polarity (+)	IntRes2 = IntRes1
01b	Negative Polarity (-)	IntRes2 = -1 XOR IntRes1
10b	Masked (+)	IntRes2 = IntRes1
11b	Masked (-)	IntRes2[i] = IntRes1[i] if reg/mem[i] invalid, else = ~IntRes1[i]

4.1.5 Output Selection

Table 4-5. Output Selection

Imm8[6]	Operation	Description
0b	Least significant index	The index returned to ECX is of the least significant set bit in IntRes2.
1b	Most significant index	The index returned to ECX is of the most significant set bit in IntRes2.

For PCMPESTRI/PCMPISTRI, the imm8 control byte bit[6] is used to determine if the index is of the least significant or most significant bit of IntRes2.

Table 4-6. Output Selection

Imm8[6]	Operation	Description
0b	Bit mask	IntRes2 is returned as the mask to the least significant bits of XMM0 with zero extension to 128 bits.
1b	Byte/word mask	IntRes2 is expanded into a byte/word mask (based on imm8[1]) and placed in XMM0. The expansion is performed by replicating each bit into all of the bits of the byte/word of the same index.

Specifically for PCMPESTRM/PCMPISTRM, the imm8 control byte bit[6] is used to determine if the mask is a 16 (8) bit mask or a 128 bit byte/word mask.

4.1.6 Valid/Invalid Override of Comparisons

PCMPxSTRx instructions allow for the possibility that an end-of-string (EOS) situation may occur within the 128-bit packed data value (see the instruction descriptions below for details). Any data elements on either source that are determined to be past the EOS are considered to be invalid, and the treatment of invalid data within a comparison pair varies depending on the aggregation function being performed.

In general, the individual comparison result for each element pair BoolRes[i.j] can be forced true or false if one or more elements in the pair are invalid. See Table 4-7.

Table 4-7. Comparison Result for Each Element Pair BoolRes[i.j]

xmm1 byte/ word	xmm2/ m128 byte/word	Imm8[3:2] = 00b (equal any)	Imm8[3:2] = 01b (ranges)	Imm8[3:2] = 10b (equal each)	Imm8[3:2] = 11b (equal ordered)
Invalid	Invalid	Force false	Force false	Force true	Force true
Invalid	Valid	Force false	Force false	Force false	Force true
Valid	Invalid	Force false	Force false	Force false	Force false
Valid	Valid	Do not force	Do not force	Do not force	Do not force

4.1.7 Summary of Im8 Control byte

Table 4-8. Summary of Imm8 Control Byte

Imm8	Description
-------0b	128-bit sources treated as 16 packed bytes.
-------1b	128-bit sources treated as 8 packed words.
------0-b	Packed bytes/words are unsigned.
------1-b	Packed bytes/words are signed.
----00--b	Mode is equal any.
----01--b	Mode is ranges.
----10--b	Mode is equal each.
----11--b	Mode is equal ordered.
---0----b	IntRes1 is unmodified.
---1----b	IntRes1 is negated (1's complement).
--0-----b	Negation of IntRes1 is for all 16 (8) bits.
--1-----b	Negation of IntRes1 is masked by reg/mem validity.
-0------b	Index of the least significant, set, bit is used (regardless of corresponding input element validity). IntRes2 is returned in least significant bits of XMM0.
-1------b	Index of the most significant, set, bit is used (regardless of corresponding input element validity). Each bit of IntRes2 is expanded to byte/word.
0-------b	This bit currently has no defined effect, should be 0.
1-------b	This bit currently has no defined effect, should be 0.

4.1.8 Diagram Comparison and Aggregation Process

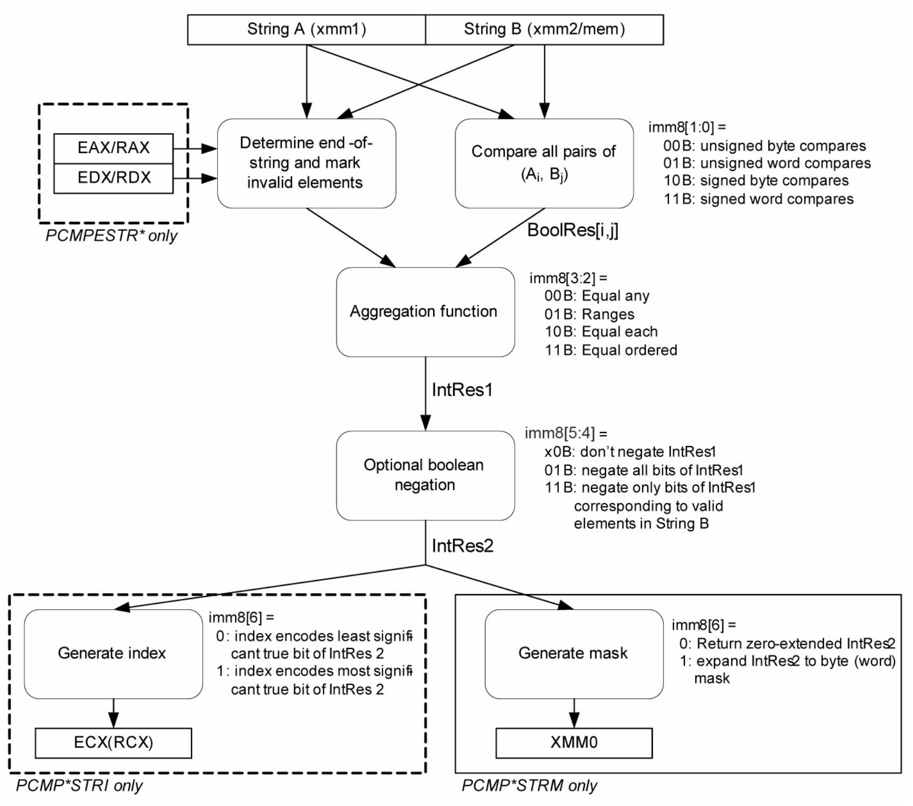

String A (xmm1)
String B (xmm2/mem)
EAX/RAX
EDX/RDX
PCMPESTR* only
Determine end-of-string and mark invalid elements
Compare all pairs of (A_i, B_j)
imm8[1:0] =
00B: unsigned byte compares
01B: unsigned word compares
10B: signed byte compares
11B: signed word compares
BoolRes[i,j]
Aggregation function
imm8[3:2] =
00B: Equal any
01B: Ranges
10B: Equal each
11B: Equal ordered
IntRes1
Optional boolean negation
imm8[5:4] =
x0B: don't negate IntRes1
01B: negate all bits of IntRes1
11B: negate only bits of IntRes1 corresponding to valid elements in String B
IntRes2
Generate index
imm8[6] =
0: index encodes least significant true bit of IntRes 2
1: index encodes most significant true bit of IntRes 2
ECX(RCX)
PCMP*STRI only
Generate mask
imm8[6] =
0: Return zero-extended IntRes2
1: expand IntRes2 to byte (word) mask
XMM0
PCMP*STRM only

Figure 4-1. Operation of PCMPSTRx and PCMPESTRx

4.2 COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1, and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- f0(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.

  f0(B,C,D) := (B AND C) XOR ((NOT(B) AND D)
- f1(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.

  f1(B,C,D) := B XOR C XOR D
- f2(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.

  f2(B,C,D) := (B AND C) XOR (B AND D) XOR (C AND D)
- f3(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as f1().

  f3(B,C,D) := B XOR C XOR D
- Ch(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).

Ch(E,F,G) := (E AND F) XOR ((NOT E) AND G)

- Maj(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).

  Maj(A,B,C) := (A AND B) XOR (A AND C) XOR (B AND C)

ROR is rotate right operation

(A ROR N) := A[N-1:0] || A[Width-1:N]

ROL is rotate left operation

(A ROL N) := A ROR (Width-N)

SHR is the right shift operation

(A SHR N) := ZEROES[N-1:0] || A[Width-1:N]

- Σ_0( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.

  Σ_0(A) := (A ROR 2) XOR (A ROR 13) XOR (A ROR 22)

- Σ_1( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.

  Σ_1(E) := (E ROR 6) XOR (E ROR 11) XOR (E ROR 25)

- σ_0( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.

  σ_0(W) := (W ROR 7) XOR (W ROR 18) XOR (W SHR 3)

- σ_1( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.

  σ_1(W) := (W ROR 17) XOR (W ROR 19) XOR (W SHR 10)

- K_i: SHA1 Constants dependent on immediate i.

  K0 = 0x5A827999

  K1 = 0x6ED9EBA1

  K2 = 0X8F1BBCDC

  K3 = 0xCA62C1D6

4.3 INSTRUCTIONS (M-U)

Chapter 4 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (M-U). See also: Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A; Chapter 4, “Instruction Set Reference, M-U,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C; and Chapter 4, “Instruction Set Reference, M-U,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

MAXPD—Maximum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double precision floating-point values between xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double precision floating-point values between xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed double precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MAX(SRC1, SRC2)
{
   IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
       ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
       ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
       ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
       ELSE DEST := SRC2;
   FI;
}
```

VMAXPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
       THEN
           IF (EVEX.b = 1) AND (SRC2 *is memory*)
               THEN
                   DEST[i+63:i] := MAX(SRC1[i+63:i], SRC2[63:0])
               ELSE
                   DEST[i+63:i] := MAX(SRC1[i+63:i], SRC2[i+63:i])
           FI;
       ELSE
           IF *merging-masking*                ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
               ELSE  DEST[i+63:i] := 0         ; zeroing-masking
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMAXPD (VEX.256 Encoded Version)

```
DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] := MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] := MAX(SRC1[255:192], SRC2[255:192])
DEST[MAXVL-1:256] := 0
```

VMAXPD (VEX.128 Encoded Version)

```
DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MAX(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] := 0
```

MAXPD (128-bit Legacy SSE Version)

```
DEST[63:0] := MAX(DEST[63:0], SRC[63:0])
DEST[127:64] := MAX(DEST[127:64], SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPD __m512d _mm512_max_pd(__m512d a, __m512d b);
VMAXPD __m512d _mm512_mask_max_pd(__m512d s, __mmask8 k, __m512d a, __m512d b,);
VMAXPD __m512d _mm512_maskz_max_pd(__mmask8 k, __m512d a, __m512d b);
VMAXPD __m512d _mm512_max_round_pd(__m512d a, __m512d b, int);
VMAXPD __m512d _mm512_mask_max_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m512d _mm512_maskz_max_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m256d _mm256_mask_max_pd(__m5256d s, __mmask8 k, __m256d a, __m256d b);
VMAXPD __m256d _mm256_maskz_max_pd(__mmask8 k, __m256d a, __m256d b);
VMAXPD __m128d _mm_mask_max_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMAXPD __m128d _mm_maskz_max_pd(__mmask8 k, __m128d a, __m128d b);
VMAXPD __m256d _mm256_max_pd (__m256d a, __m256d b);
(V)MAXPD __m128d _mm_max_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-48, "Type E2 Class Exception Conditions."

MAXPS—Maximum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5F /r MAXPS xmm1, xmm2/m128	A	V/V	SSE	Return the maximum single precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the maximum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed single precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMAXPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := MAX(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] := MAX(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                  ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0           ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMAXPS (VEX.256 Encoded Version)

```
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] := MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] := MAX(SRC1[191:160], SRC2[191:160])
DEST[223:192] := MAX(SRC1[223:192], SRC2[223:192])
DEST[255:224] := MAX(SRC1[255:224], SRC2[255:224])
DEST[MAXVL-1:256] := 0
```

VMAXPS (VEX.128 Encoded Version)

```
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MAX(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] := 0
```

MAXPS (128-bit Legacy SSE Version)

```
DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[63:32] := MAX(DEST[63:32], SRC[63:32])
DEST[95:64] := MAX(DEST[95:64], SRC[95:64])
DEST[127:96] := MAX(DEST[127:96], SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXPS __m512 _mm512_max_ps( __m512 a, __m512 b);
VMAXPS __m512 _mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_maskz_max_ps( __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_max_round_ps( __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_maskz_max_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m256 _mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMAXPS __m256 _mm256_maskz_max_ps( __mmask8 k, __m256 a, __m256 b);
VMAXPS __m128 _mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMAXPS __m128 _mm_maskz_max_ps( __mmask8 k, __m128 a, __m128 b);
VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 b);
MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MAXSD—Return Maximum Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	A	V/V	SSE2	Return the maximum scalar double precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.
EVEX.LLIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Compares the low double precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMAXSD (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMAXSD (VEX.128 Encoded Version)

```
DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MAXSD (128-bit Legacy SSE Version)

```
DEST[63:0] := MAX(DEST[63:0], SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSD __m128d _mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

MAXSS—Return Maximum Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	A	V/V	SSE	Return the maximum scalar single precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.
EVEX.LLIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1	Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Compares the low single precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMAXSS (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMAXSS (VEX.128 Encoded Version)

```
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MAXSS (128-bit Legacy SSE Version)

```
DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSS __m128 _mm_max_round_ss( __m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-49, "Type E3 Class Exception Conditions."

MINPD—Minimum of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double precision floating-point values between xmm1 and xmm2/mem
VEX.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double precision floating-point values between xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed double precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMINPD (EVEX Encoded Version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := MIN(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] := MIN(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMINPD (VEX.256 Encoded Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])
DEST[191:128] := MIN(SRC1[191:128], SRC2[191:128])
DEST[255:192] := MIN(SRC1[255:192], SRC2[255:192])
```

VMINPD (VEX.128 Encoded Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] := 0
```

MINPD (128-bit Legacy SSE Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := MIN(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPD __m512d _mm512_min_pd( __m512d a, __m512d b);
VMINPD __m512d _mm512_mask_min_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_maskz_min_pd( __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_min_round_pd( __m512d a, __m512d b, int);
VMINPD __m512d _mm512_mask_min_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d _mm512_maskz_min_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m256d _mm256_mask_min_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d _mm256_maskz_min_pd( __mmask8 k, __m256d a, __m256d b);
VMINPD __m128d _mm_mask_min_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d _mm_maskz_min_pd( __mmask8 k, __m128d a, __m128d b);
VMINPD __m256d _mm256_min_pd (__m256d a, __m256d b);
MINPD __m128d _mm_min_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MINPS—Minimum of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Return the minimum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed single precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

VMINPS (EVEX Encoded Version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := MIN(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] := MIN(SRC1[i+31:i], SRC2[i+31:i])
            FI;
            ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMINPS (VEX.256 Encoded Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] := MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] := MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] := MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] := MIN(SRC1[255:224], SRC2[255:224])
```

VMINPS (VEX.128 Encoded Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] := 0
```

MINPS (128-bit Legacy SSE Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPS __m512 _mm512_min_ps( __m512 a, __m512 b);
VMINPS __m512 _mm512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_maskz_min_ps( __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_min_round_ps( __m512 a, __m512 b, int);
VMINPS __m512 _mm512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m512 _mm512_maskz_min_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m256 _mm256_mask_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMINPS __m256 _mm256_maskz_min_ps( __mmask8 k, __m256 a, __m25 b);
VMINPS __m128 _mm_mask_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMINPS __m128 _mm_maskz_min_ps( __mmask8 k, __m128 a, __m128 b);
VMINPS __m256 _mm256_min_ps (__m256 a, __m256 b);
MINPS __m128 _mm_min_ps (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MINSD—Return Minimum Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSD xmm1, xmm2/m64	A	V/V	SSE2	Return the minimum scalar double precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.
EVEX.LLIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Compares the low double precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VMINSD is encoded with VEX.L=0. Encoding VMINSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
         ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
         ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
         ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
         ELSE DEST := SRC2;
    FI;
}
```

MINSD (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN      DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
    ELSE
         IF *merging-masking*                  ; merging-masking
              THEN *DEST[63:0] remains unchanged*
              ELSE                             ; zeroing-masking
                   THEN DEST[63:0] := 0
         FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MINSD (VEX.128 Encoded Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MINSD (128-bit Legacy SSE Version)

```
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSD __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d _mm_maskz_min_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d _mm_min_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

MINSS—Return Minimum Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	A	V/V	SSE	Return the minimum scalar single precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.
EVEX.LLIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F OR AVX10.1	Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Compares the low single precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
        ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
        ELSE DEST := SRC2;
    FI;
}
```

MINSS (EVEX Encoded Version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMINSS (VEX.128 Encoded Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MINSS (128-bit Legacy SSE Version)

```
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSS __m128 _mm_min_round_ss( __m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8[1], r8[1]	MR	Valid	Valid	Move r8 to r/m8.
89 /r	MOV r/m16, r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32, r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64, r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8[1], r/m8[1]	RM	Valid	Valid	Move r/m8 to r8.
8B /r	MOV r16, r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32, r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64, r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16, Sreg[2]	MR	Valid	Valid	Move segment register to r/m16.
8C /r	MOV r16/r32/m16, Sreg[2]	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/m16.
REX.W + 8C /r	MOV r64/m16, Sreg[2]	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
8E /r	MOV Sreg, r/m16[2]	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg, r/m64[2]	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL, moffs8[3]	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL, moffs8[3]	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX, moffs16[3]	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX, moffs32[3]	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX, moffs64[3]	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8, AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8[1], AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16[3], AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32[3], EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64[3], RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8[1], imm8	OI	Valid	Valid	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /0 ib	MOV r/m8[1], imm8	MI	Valid	Valid	Move imm8 to r/m8.
C7 /0 iw	MOV r/m16, imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /0 id	MOV r/m32, imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /0 id	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following “Description” section for further information).
3. The moffs8, moffs16, moffs32, and moffs64 operands specify a simple offset relative to the segment base, where 8, 16, 32, and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32, or 64 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
FD	AL/AX/EAX/RAX	Moffs	N/A	N/A
TD	Moffs (w)	AL/AX/EAX/RAX	N/A	N/A
OI	opcode + rd (w)	imm8/16/32/64	N/A	N/A
MI	ModRM:r/m (w)	imm8/16/32/64	N/A	N/A

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the "Operation" algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an event can be delivered. See Section 7.8.3, "Masking Exceptions and Interrupts When Switching Stacks," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

When executing MOV Reg, Sreg, the processor copies the content of Sreg to the 16 least significant bits of the general-purpose register. The upper bits of the destination register are zero for most IA-32 processors (Pentium Pro processors and later) and all Intel 64 processors, with the exception that bits 31:16 are undefined for Intel Quark X1000 processors, Pentium, and earlier processors.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := SRC;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

```
IF SS is loaded
    THEN
        IF segment selector is NULL
            THEN #GP(0); FI;
        IF segment selector index is outside descriptor table limits
        OR segment selector's RPL ≠ CPL
```

```
            OR segment is not a writable data segment
            OR DPL ≠ CPL
                THEN #GP(selector); FI;
            IF segment not marked present
                THEN #SS(selector);
                ELSE
                    SS := segment selector;
                    SS := segment descriptor; FI;
FI;
IF DS, ES, FS, or GS is loaded with non-NULL selector
THEN
    IF segment selector index is outside descriptor table limits
    OR segment is not a data or readable code segment
    OR ((segment is a data or nonconforming code segment) AND ((RPL > DPL) or (CPL > DPL)))
        THEN #GP(selector); FI;
    IF segment not marked present
        THEN #NP(selector);
        ELSE
            SegmentRegister := segment selector;
            SegmentRegister := segment descriptor; FI;
FI;
IF DS, ES, FS, or GS is loaded with NULL selector
    THEN
        SegmentRegister := segment selector;
        SegmentRegister := segment descriptor;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with NULL segment selector.
	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, and either the RPL or the CPL is greater than the DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If an attempt is made to load SS register with NULL segment selector when CPL = 3.
	If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the memory access to the descriptor table is non-canonical.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the stack address is in a non-canonical form.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

MOVAPD—Move Aligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed double precision floating-point values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed double precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves 2, 4 or 8 double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register destination are zeroed.

Operation

VMOVAPD (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVAPD (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
        ELSE *DEST[i+63:i] remains unchanged*           ; merging-masking

    FI;
ENDFOR;
```

VMOVAPD (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0                 ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVAPD (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVAPD (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVAPD (VEX.128 Encoded Version, Load - and Register Copy)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

MOVAPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPD (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m512d _mm512_load_pd(void * m);
VMOVAPD __m512d _mm512_mask_load_pd(__m512d s, __mmask8 k, void * m);
VMOVAPD __m512d _mm512_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm512_store_pd(void * d, __m512d a);
VMOVAPD void _mm512_mask_store_pd(void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d _mm256_mask_load_pd(__m256d s, __mmask8 k, void * m);
VMOVAPD __m256d _mm256_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm256_mask_store_pd(void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d _mm_mask_load_pd(__m128d s, __mmask8 k, void * m);
VMOVAPD __m128d _mm_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm_mask_store_pd(void * d, __mmask8 k, __m128d a);
MOVAPD __m256d _mm256_load_pd (double * p);
MOVAPD void _mm256_store_pd(double * p, __m256d a);
MOVAPD __m128d _mm_load_pd (double * p);
MOVAPD void _mm_store_pd(double * p, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E1 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVAPS—Move Aligned Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 28 /r MOVAPS xmm1, xmm2/m128	A	V/V	SSE	Move aligned packed single precision floating-point values from xmm2/mem to xmm1.
NP 0F 29 /r MOVAPS xmm2/m128, xmm1	B	V/V	SSE	Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed single precision floating-point values from xmm2/mem to xmm1.
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed single precision floating-point values from ymm2/mem to ymm1.
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed single precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves 4, 8 or 16 single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:

Moves 256 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:

Moves 128 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

Operation

VMOVAPS (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVAPS (EVEX Encoded Versions, Store Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVAPS (EVEX Encoded Versions, Load Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVAPS (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVAPS (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVAPS (VEX.128 Encoded Version, Load - and Register Copy)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

MOVAPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPS (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPS __m512 _mm512_load_ps(void * m);
VMOVAPS __m512 _mm512_mask_load_ps(__m512 s, __mmask16 k, void * m);
VMOVAPS __m512 _mm512_maskz_load_ps(__mmask16 k, void * m);
VMOVAPS void _mm512_store_ps(void * d, __m512 a);
VMOVAPS void _mm512_mask_store_ps(void * d, __mmask16 k, __m512 a);
VMOVAPS __m256 _mm256_mask_load_ps(__m256 a, __mmask8 k, void * s);
VMOVAPS __m256 _mm256_maskz_load_ps(__mmask8 k, void * s);
VMOVAPS void _mm256_mask_store_ps(void * d, __mmask8 k, __m256 a);
VMOVAPS __m128 _mm_mask_load_ps(__m128 a, __mmask8 k, void * s);
VMOVAPS __m128 _mm_maskz_load_ps(__mmask8 k, void * s);
VMOVAPS void _mm_mask_store_ps(void * d, __mmask8 k, __m128 a);
MOVAPS __m256 _mm256_load_ps (float * p);
MOVAPS void _mm256_store_ps(float * p, __m256 a);
MOVAPS __m128 _mm_load_ps (float * p);
MOVAPS void _mm_store_ps(float * p, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, “Type 1 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-46, “Type E1 Class Exception Conditions.”

MOVDDUP—Replicate Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	A	V/V	SSE3	Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	A	V/V	AVX	Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index double precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move double precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move even index double precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1	Move even index double precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	MOVDDUP	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

For 256-bit or higher versions: Duplicates even-indexed double precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the write-mask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

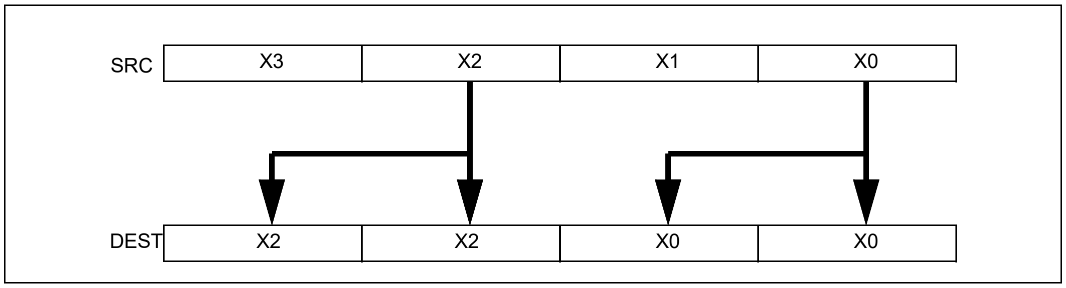


Figure 4-2. VMOVDDUP Operation

Operation

VMOVDDUP (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_SRC[63:0] := SRC[63:0]
TMP_SRC[127:64] := SRC[63:0]
IF VL >= 256
    TMP_SRC[191:128] := SRC[191:128]
    TMP_SRC[255:192] := SRC[191:128]
FI;
IF VL >= 512
    TMP_SRC[319:256] := SRC[319:256]
    TMP_SRC[383:320] := SRC[319:256]
    TMP_SRC[477:384] := SRC[477:384]
    TMP_SRC[511:484] := SRC[477:384]
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0               ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDDUP (VEX.256 Encoded Version)

```
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[191:128] := SRC[191:128]
DEST[255:192] := SRC[191:128]
DEST[MAXVL-1:256] := 0
```

VMOVDDUP (VEX.128 Encoded Version)

```
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] := 0
```

MOVDDUP (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVDDUP __m512d _mm512_movedup_pd( __m512d a);
VMOVDDUP __m512d _mm512_mask_movedup_pd(__m512d s, __mmask8 k, __m512d a);
VMOVDDUP __m512d _mm512_maskz_movedup_pd( __mmask8 k, __m512d a);
VMOVDDUP __m256d _mm256_mask_movedup_pd(__m256d s, __mmask8 k, __m256d a);
VMOVDDUP __m256d _mm256_maskz_movedup_pd( __mmask8 k, __m256d a);
VMOVDDUP __m128d _mm_mask_movedup_pd(__m128d s, __mmask8 k, __m128d a);
VMOVDDUP __m128d _mm_maskz_movedup_pd( __mmask8 k, __m128d a);
MOVDDUP __m256d _mm256_movedup_pd (__m256d a);
MOVDDUP __m128d _mm_movedup_pd (__m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-54, “Type E5NF Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6E /r MOVD mm, r/m32	A	V/V	MMX	Move doubleword from r/m32 to mm.
NP REX.W + 0F 6E /r MOVQ mm, r/m64	A	V/N.E.	MMX	Move quadword from r/m64 to mm.
NP 0F 7E /r MOVD r/m32, mm	B	V/V	MMX	Move doubleword from mm to r/m32.
NP REX.W + 0F 7E /r MOVQ r/m64, mm	B	V/N.E.	MMX	Move quadword from mm to r/m64.
66 0F 6E /r MOVD xmm, r/m32	A	V/V	SSE2	Move doubleword from r/m32 to xmm.
66 REX.W 0F 6E /r MOVQ xmm, r/m64	A	V/N.E.	SSE2	Move quadword from r/m64 to xmm.
66 0F 7E /r MOVD r/m32, xmm	B	V/V	SSE2	Move doubleword from xmm register to r/m32.
66 REX.W 0F 7E /r MOVQ r/m64, xmm	B	V/N.E.	SSE2	Move quadword from xmm register to r/m64.
VEX.128.66.0F.W0 6E / VMOVD xmm1, r32/m32	A	V/V	AVX	Move doubleword from r/m32 to xmm1.
VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	A	V/N.E.[1]	AVX	Move quadword from r/m64 to xmm1.
VEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	B	V/V	AVX	Move doubleword from xmm1 register to r/m32.
VEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	B	V/N.E.[1]	AVX	Move quadword from xmm1 register to r/m64.
EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	C	V/V	AVX512F OR AVX10.1	Move doubleword from r/m32 to xmm1.
EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	C	V/N.E.	AVX512F OR AVX10.1	Move quadword from r/m64 to xmm1.
EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	D	V/V	AVX512F OR AVX10.1	Move doubleword from xmm1 register to r/m32.
EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	D	V/N.E.[1]	AVX512F OR AVX10.1	Move quadword from xmm1 register to r/m64.

NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.B prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVD (When Destination Operand is an MMX Technology Register)

```
    DEST[31:0] := SRC;
    DEST[63:32] := 00000000H;
```

MOVD (When Destination Operand is an XMM Register)

```
    DEST[31:0] := SRC;
    DEST[127:32] := 000000000000000000000000H;
    DEST[MAXVL-1:128] (Unmodified)
```

MOVD (When Source Operand is an MMX Technology or XMM Register)

```
    DEST := SRC[31:0];
```

VMOVD (VEX-Encoded Version when Destination is an XMM Register)

```
    DEST[31:0] := SRC[31:0]
    DEST[MAXVL-1:32] := 0
```

MOVQ (When Destination Operand is an XMM Register)

```
    DEST[63:0] := SRC[63:0];
    DEST[127:64] := 0000000000000000H;
    DEST[MAXVL-1:128] (Unmodified)
```

MOVQ (When Destination Operand is r/m64)

```
    DEST[63:0] := SRC[63:0];
```

MOVQ (When Source Operand is an XMM Register or r/m64)

```
    DEST := SRC[63:0];
```

VMOVQ (VEX-Encoded Version When Destination is an XMM Register)

```
    DEST[63:0] := SRC[63:0]
    DEST[MAXVL-1:64] := 0
```

VMOVD (EVEX-Encoded Version When Destination is an XMM Register)

```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0
```

VMOVQ (EVEX-Encoded Version When Destination is an XMM Register)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVD __m64 _mm_cvtsi32_si64 (int i )
MOVD int _mm_cvtsi64_si32 ( __m64m )
MOVD __m128i _mm_cvtsi32_si128 (int a)
MOVD int _mm_cvtsi128_si32 ( __m128i a)
MOVQ __int64 _mm_cvtsi128_si64(__m128i);
MOVQ __m128i _mm_cvtsi64_si128(__int64);
VMOVD __m128i _mm_cvtsi32_si128( int);
VMOVD int _mm_cvtsi128_si32( __m128i );
VMOVQ __m128i _mm_cvtsi64_si128 (__int64);
VMOVQ __int64 _mm_cvtsi128_si64(__m128i );
VMOVQ __m128i _mm_loadl_epi64( __m128i * s);
VMOVQ void _mm_storel_epi64( __m128i * d, __m128i s);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVDQA,VMOVDQA32/64—Move Aligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQA32 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQA32 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA32 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQA64 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0          ; zeroing-masking
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VMOVDQA64 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*           ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA64 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0                 ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQA (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVDQA (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVDQA (VEX.128 Encoded Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

VMOVDQA (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQA (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQA32 __m512i _mm512_load_epi32(void * sa);
VMOVDQA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i _mm512_maskz_load_epi32(__mmask16 k, void * sa);
VMOVDQA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i _mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i _mm256_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i _mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i _mm_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void _mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i _mm512_load_epi64(void * sa);
VMOVDQA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i _mm512_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i _mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i _mm256_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void _mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i _mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i _mm_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void _mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i _mm256_load_si256 (__m256i * p);
MOVDQA _mm256_store_si256(_m256i *p, __m256i a);
MOVDQA __m128i _mm_load_si128 (__m128i * p);
MOVDQA void _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E1 Class Exception Conditions.”
Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVDQU,VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW OR AVX10.1	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW OR AVX10.1	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW OR AVX10.1	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW OR AVX10.1	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/mm128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQU8 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] := 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU8 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] :=
            SRC[i+7:i]
        ELSE *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU8 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] := 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU16 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU16 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            SRC[i+15:i]
        ELSE *DEST[i+15:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU16 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU32 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU32 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU32 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VMOVDQU64 (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU64 (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*       ; merging-masking

    FI;
ENDFOR;
```

VMOVDQU64 (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVDQU (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVDQU (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

```
VMOVDQU (VEX.128 encoded version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

VMOVDQU (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQU (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVDQU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVDQU16 __m512i _mm512_maskz_loadu_epi16( __mmask32 k, void * sa);
VMOVDQU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVDQU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVDQU16 __m256i _mm256_maskz_loadu_epi16( __mmask16 k, void * sa);
VMOVDQU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVDQU16 __m128i _mm_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVDQU16 __m128i _mm_maskz_loadu_epi16( __mmask8 k, void * sa);
VMOVDQU16 void _mm_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVDQU32 __m512i _mm512_loadu_epi32( void * sa);
VMOVDQU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQU32 __m512i _mm512_maskz_loadu_epi32( __mmask16 k, void * sa);
VMOVDQU32 void _mm512_storeu_epi32(void * d, __m512i a);
VMOVDQU32 void _mm512_mask_storeu_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQU32 __m256i _mm256_maskz_loadu_epi32( __mmask8 k, void * sa);
VMOVDQU32 void _mm256_storeu_epi32(void * d, __m256i a);
VMOVDQU32 void _mm256_mask_storeu_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQU32 __m128i _mm_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQU32 __m128i _mm_maskz_loadu_epi32( __mmask8 k, void * sa);
VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64( void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8( __mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8( __mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8( __mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256 (__m256i * p);
```

MOVDQU _mm256_storeu_si256(_m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128 (__m128i * p);
MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVHLPS—Move Packed Single Precision Floating-Point Values High to Low

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVHLPS xmm1, xmm2	RM	V/V	SSE	Move two packed single precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r) / EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

128-bit and EVEX three-argument form:

Moves two packed single precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHLPS (128-bit Two-Argument Form)

```
DEST[63:0] := SRC[127:64]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVHLPS (128-bit Three-Argument Form - VEX & EVEX)

```
DEST[63:0] := SRC2[127:64]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B required.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-57, “Type E7NM Class Exception Conditions.”

MOVHPD—Move High Packed Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	A	V/V	SSE2	Move double precision floating-point value from m64 to high quadword of xmm1.
VEX.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	B	V/V	AVX	Merge double precision floating-point value from m64 and the low quadword of xmm1.
EVEX.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge double precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17 /r MOVHPD m64, xmm1	C	V/V	SSE2	Move double precision floating-point value from high quadword of xmm1 to m64.
VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1	C	V/V	AVX	Move double precision floating-point value from high quadword of xmm1 to m64.
EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move double precision floating-point value from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHPD (128-bit Legacy SSE Load)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPD (VEX.128 & EVEX Encoded Load)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

VMOVHPD (Store)

```
DEST[63:0] := SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)
MOVHPD void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

MOVHPS—Move High Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVHPS xmm1, m64	A	V/V	SSE	Move two packed single precision floating-point values from m64 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.
EVEX.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.
NP 0F 17 /r MOVHPS m64, xmm1	C	V/V	SSE	Move two packed single precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	C	V/V	AVX	Move two packed single precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move two packed single precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two single precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores two packed single precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE Load)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPS (VEX.128 and EVEX Encoded Load)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

VMOVHPS (Store)

```
DEST[63:0] := SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128 _mm_loadh_pi (__m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, "Type E9NF Class Exception Conditions."

MOVLHPS—Move Packed Single Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
RVM	ModRM:reg (w)	VEX.vvvv (r) / EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLHPS (128-bit Two-Argument Form)

```
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVLHPS (128-bit Three-Argument Form - VEX & EVEX)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-57, “Type E7NM Class Exception Conditions.”

MOVLPD—Move Low Packed Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	A	V/V	SSE2	Move double precision floating-point value from m64 to low quadword of xmm1.
VEX.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double precision floating-point value from m64 and the high quadword of xmm1.
EVEX.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge double precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	C	V/V	SSE2	Move double precision floating-point value from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	C	V/V	AVX	Move double precision floating-point value from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move double precision floating-point value from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLPD (128-bit Legacy SSE Load)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPD (VEX.128 & EVEX Encoded Load)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVLPD (Store)

```
DEST[63:0] := SRC[63:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)
MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

MOVLPS—Move Low Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVLPS xmm1, m64	A	V/V	SSE	Move two packed single precision floating-point values from m64 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.
EVEX.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	D	V/V	AVX512F OR AVX10.1	Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	C	V/V	SSE	Move two packed single precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	C	V/V	AVX	Move two packed single precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	E	V/V	AVX512F OR AVX10.1	Move two packed single precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two packed single precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Loads two packed single precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLPS (128-bit Legacy SSE Load)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPS (VEX.128 & EVEX Encoded Load)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVLPS (Store)

```
DEST[63:0] := SRC[63:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	A	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	A	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	A	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed integer values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed integer values in zmm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	B	V/V	AVX512F OR AVX10.1	Move packed integer values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTDQ(EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
```

1. ModRM.MOD != 011B

```
DEST[MAXVL-1:VL] := 0
```

MOVNTDQ (Legacy and VEX Versions)

```
DEST := SRC
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVNTDQ void _mm512_stream_si512(void * p, __m512i a);
VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);
MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-47, “Type E1NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA ymm1, m256	A	V/V	AVX2	Move 256-bit data from m256 to ymm1 using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move 128-bit data from m128 to xmm1 using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move 256-bit data from m256 to ymm1 using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	B	V/V	AVX512F OR AVX10.1	Move 512-bit data from m512 to zmm1 using non-temporal hint if WC memory type.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the non-temporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by

1. ModRM.MOD != 011B

other agents in the system. A processor's implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

Operation

MOVNTDQA (128bit- Legacy SSE Form)

```
DEST := SRC
DEST[MAXVL-1:128] (Unmodified)
```

VMOVNTDQA (VEX.128 and EVEX.128 Encoded Form)

```
DEST := SRC
DEST[MAXVL-1:128] := 0
```

VMOVNTDQA (VEX.256 and EVEX.256 Encoded Forms)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVNTDQA (EVEX.512 Encoded Form)

```
DEST[511:0] := SRC[511:0]
DEST[MAXVL-1:512] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVNTDQA __m512i _mm512_stream_load_si512(__m512i const* p);
MOVNTDQA __m128i _mm_stream_load_si128 (const __m128i *p);
VMOVNTDQA __m256i _mm256_stream_load_si256 (__m256i const* p);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-18, "Type 1 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-47, "Type E1NF Class Exception Conditions."

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTPD—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	A	V/V	SSE2	Move packed double precision values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	A	V/V	AVX	Move packed double precision values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	A	V/V	AVX	Move packed double precision values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed double precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed double precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	B	V/V	AVX512F OR AVX10.1	Move packed double precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the packed double precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTPD (EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
```

1. ModRM.MOD != 011B

```
DEST[MAXVL-1:VL] := 0
```

MOVNTPD (Legacy and VEX Versions)

```
DEST := SRC
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void _mm512_stream_pd(double * p, __m512d a);
VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);
MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E1NF Class Exception Conditions.”
Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2B /r MOVNTPS m128, xmm1	A	V/V	SSE	Move packed single precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	A	V/V	AVX	Move packed single precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	A	V/V	AVX	Move packed single precision values ymm1 to mem using non-temporal hint.
EVEX.128.0F.W0 2B /r VMOVNTPS m128, xmm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed single precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.0F.W0 2B /r VMOVNTPS m256, ymm1	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move packed single precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.0F.W0 2B /r VMOVNTPS m512, zmm1	B	V/V	AVX512F OR AVX10.1	Move packed single precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves the packed single precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VMOVNTPS (EVEX Encoded Versions)

```
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
DEST[MAXVL-1:VL] := 0
```

1. ModRM.MOD != 011B

```
MOVNTPS
DEST := SRC
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);
MOVNTPS void _mm_stream_ps (float * p, __m128d a);
VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, “Type 1 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-47, “Type E1NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /r MOVQ mm, mm/m64	A	V/V	MMX	Move quadword from mm/m64 to mm.
NP 0F 7F /r MOVQ mm/m64, mm	B	V/V	MMX	Move quadword from mm to mm/m64.
F3 0F 7E /r MOVQ xmm1, xmm2/m64	A	V/V	SSE2	Move quadword from xmm2/mem64 to xmm1.
VEX.128.F3.0F.WIG 7E /r VMOVQ xmm1, xmm2/m64	A	V/V	AVX	Move quadword from xmm2 to xmm1.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2/m64	C	V/V	AVX512F OR AVX10.1	Move quadword from xmm2/m64 to xmm1.
66 0F D6 /r MOVQ xmm2/m64, xmm1	B	V/V	SSE2	Move quadword from xmm1 to xmm2/mem64.
VEX.128.66.0F.WIG D6 /r VMOVQ xmm1/m64, xmm2	B	V/V	AVX	Move quadword from xmm2 register to xmm1/m64.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	D	V/V	AVX512F OR AVX10.1	Move quadword from xmm2 register to xmm1/m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVQ Instruction When Operating on MMX Technology Registers and Memory Locations

```
    DEST := SRC;
```

MOVQ Instruction When Source and Destination Operands are XMM Registers

```
    DEST[63:0] := SRC[63:0];
    DEST[127:64] := 0000000000000000H;
```

MOVQ Instruction When Source Operand is XMM Register and Destination
```
operand is memory location:
    DEST := SRC[63:0];
```

MOVQ Instruction When Source Operand is Memory Location and Destination
```
operand is XMM register:
    DEST[63:0] := SRC;
    DEST[127:64] := 0000000000000000H;
```

VMOVQ (VEX.128.F3.0F 7E) With XMM Register Source and Destination
```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (VEX.128.66.0F D6) With XMM Register Source and Destination
```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (7E - EVEX Encoded Version) With XMM Register Source and Destination
```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (D6 - EVEX Encoded Version) With XMM Register Source and Destination
```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (7E) With Memory Source
```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

VMOVQ (7E - EVEX Encoded Version) With Memory Source
```
DEST[63:0] := SRC[63:0]
DEST[:MAXVL-1:64] := 0
```

VMOVQ (D6) With Memory DEST
```
DEST[63:0] := SRC2[63:0]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVQ __m128i _mm_loadu_si64( void * s);
VMOVQ void _mm_storeu_si64( void * d, __m128i s);
MOVQ m128i _mm_move_epi64(__m128i a)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 24-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

MOVSD—Move or Merge Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	A	V/V	SSE2	Move scalar double precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	A	V/V	SSE2	Load scalar double precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	C	V/V	SSE2	Move scalar double precision floating-point value from xmm2 register to xmm1/m64.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar double precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, m64	D	V/V	AVX	Load scalar double precision floating-point value from m64 to xmm1 register.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	E	V/V	AVX	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD m64, xmm1	C	V/V	AVX	Store scalar double precision floating-point value from xmm1 register to m64.
EVEX.LLIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F OR AVX10.1	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LLIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	F	V/V	AVX512F OR AVX10.1	Load scalar double precision floating-point value from m64 to xmm1 register under writemask k1.
EVEX.LLIG.F2.0F.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F OR AVX10.1	Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LLIG.F2.0F.W1 11 /r VMOVSD m64 {k1}, xmm1	G	V/V	AVX512F OR AVX10.1	Store scalar double precision floating-point value from xmm1 register to m64 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	N/A	ModRM:r/m (w)	EVEX.vvvv (r)	ModRM:reg (r)	N/A
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves a scalar double precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

Operation

VMOVSD (EVEX.LLIG.F2.0F 10 /r: VMOVSD xmm1, m64 With Support for 32 Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[MAXVL-1:64] := 0
```

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD m64, xmm1 With Support for 32 Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC[63:0]
    ELSE    *DEST[63:0] remains unchanged*          ; merging-masking
FI;
```

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, xmm2)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVSD (VEX.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)

```
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, m64)

```
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0
```

MOVSD/VMOVSD (128-bit Versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)

```
DEST[63:0] := SRC[63:0]
```

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, m64)

```
DEST[63:0] := SRC[63:0]
DEST[127:64] := 0
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSD __m128d _mm_mask_load_sd(__m128d s, __mmask8 k, double * p);
VMOVSD __m128d _mm_maskz_load_sd( __mmask8 k, double * p);
VMOVSD __m128d _mm_mask_move_sd(__m128d sh, __mmask8 k, __m128d sl, __m128d a);
VMOVSD __m128d _mm_maskz_move_sd( __mmask8 k, __m128d s, __m128d a);
VMOVSD void _mm_mask_store_sd(double * p, __mmask8 k, __m128d s);
MOVSD __m128d _mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d _mm_move_sd ( __m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-60, “Type E10 Class Exception Conditions.”

MOVSHDUP—Replicate Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	A	V/V	SSE3	Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	A	V/V	AVX	Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	A	V/V	AVX	Move odd index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move odd index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move odd index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1	Move odd index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Duplicates odd-indexed single precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

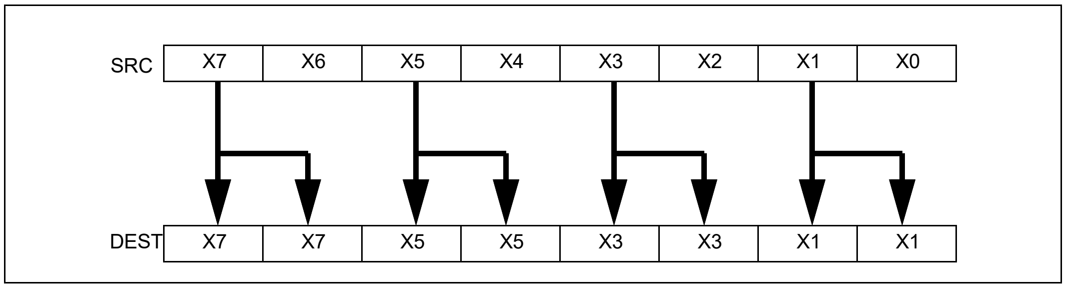


Figure 4-3. MOVSHDUP Operation

Operation

VMOVSHDUP (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] := SRC[63:32]
TMP_SRC[63:32] := SRC[63:32]
TMP_SRC[95:64] := SRC[127:96]
TMP_SRC[127:96] := SRC[127:96]
IF VL >= 256
    TMP_SRC[159:128] := SRC[191:160]
    TMP_SRC[191:160] := SRC[191:160]
    TMP_SRC[223:192] := SRC[255:224]
    TMP_SRC[255:224] := SRC[255:224]
FI;
IF VL >= 512
    TMP_SRC[287:256] := SRC[319:288]
    TMP_SRC[319:288] := SRC[319:288]
    TMP_SRC[351:320] := SRC[383:352]
    TMP_SRC[383:352] := SRC[383:352]
    TMP_SRC[415:384] := SRC[447:416]
    TMP_SRC[447:416] := SRC[447:416]
    TMP_SRC[479:448] := SRC[511:480]
    TMP_SRC[511:480] := SRC[511:480]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVSHDUP (VEX.256 Encoded Version)

```
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[159:128] := SRC[191:160]
DEST[191:160] := SRC[191:160]
DEST[223:192] := SRC[255:224]
DEST[255:224] := SRC[255:224]
DEST[MAXVL-1:256] := 0
```

VMOVSHDUP (VEX.128 Encoded Version)

```
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[MAXVL-1:128] := 0
```

MOVSHDUP (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC[63:32]
DEST[63:32] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[127:96] := SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSHDUP __m512 _mm512_movehdup_ps( __m512 a);
VMOVSHDUP __m512 _mm512_mask_movehdup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSHDUP __m512 _mm512_maskz_movehdup_ps( __mmask16 k, __m512 a);
VMOVSHDUP __m256 _mm256_mask_movehdup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSHDUP __m256 _mm256_maskz_movehdup_ps( __mmask8 k, __m256 a);
VMOVSHDUP __m128 _mm_mask_movehdup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSHDUP __m128 _mm_maskz_movehdup_ps( __mmask8 k, __m128 a);
VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);
VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVSLDUP—Replicate Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	A	V/V	AVX	Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move even index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move even index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F OR AVX10.1	Move even index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Duplicates even-indexed single precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

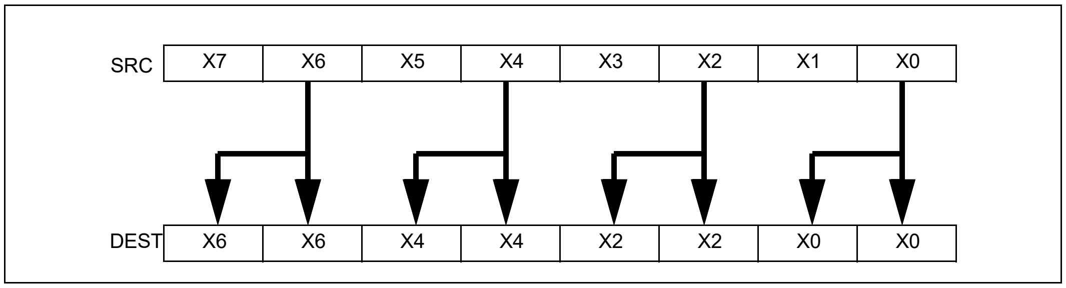


Figure 4-4. MOVSLDUP Operation

Operation

VMOVSLDUP (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] := SRC[31:0]
TMP_SRC[63:32] := SRC[31:0]
TMP_SRC[95:64] := SRC[95:64]
TMP_SRC[127:96] := SRC[95:64]
IF VL >= 256
    TMP_SRC[159:128] := SRC[159:128]
    TMP_SRC[191:160] := SRC[159:128]
    TMP_SRC[223:192] := SRC[223:192]
    TMP_SRC[255:224] := SRC[223:192]
FI;
IF VL >= 512
    TMP_SRC[287:256] := SRC[287:256]
    TMP_SRC[319:288] := SRC[287:256]
    TMP_SRC[351:320] := SRC[351:320]
    TMP_SRC[383:352] := SRC[351:320]
    TMP_SRC[415:384] := SRC[415:384]
    TMP_SRC[447:416] := SRC[415:384]
    TMP_SRC[479:448] := SRC[479:448]
    TMP_SRC[511:480] := SRC[479:448]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVSLDUP (VEX.256 Encoded Version)

```
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[159:128] := SRC[159:128]
DEST[191:160] := SRC[159:128]
DEST[223:192] := SRC[223:192]
DEST[255:224] := SRC[223:192]
DEST[MAXVL-1:256] := 0
```

VMOVSLDUP (VEX.128 Encoded Version)

```
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] := 0
```

MOVSLDUP (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSLDUP __m512 _mm512_moveldup_ps( __m512 a);
VMOVSLDUP __m512 _mm512_mask_moveldup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSLDUP __m512 _mm512_maskz_moveldup_ps( __mmask16 k, __m512 a);
VMOVSLDUP __m256 _mm256_mask_moveldup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSLDUP __m256 _mm256_maskz_moveldup_ps( __mmask8 k, __m256 a);
VMOVSLDUP __m128 _mm_mask_moveldup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSLDUP __m128 _mm_maskz_moveldup_ps( __mmask8 k, __m128 a);
VMOVSLDUP __m256 _mm256_moveldup_ps (__m256 a);
VMOVSLDUP __m128 _mm_moveldup_ps (__m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVSS—Move or Merge Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single precision floating-point value from m32 to xmm1 register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	C	V/V	SSE	Move scalar single precision floating-point value from xmm1 register to xmm2/m32.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	E	V/V	AVX	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	C	V/V	AVX	Move scalar single precision floating-point value from xmm1 register to m32.
EVEX.LLIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LLIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	F	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.LLIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LLIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	G	V/V	AVX512F OR AVX10.1	Move scalar single precision floating-point values from xmm1 to m32 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
D	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	N/A	ModRM:r/m (w)	EVEX.vvvv (r)	ModRM:reg (r)	N/A
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Moves a scalar single precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction "VMOVSS m32, xmm1", VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction "VMOVSS mv {k1}, xmm1", EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMOVSS (EVEX.LLIG.F3.0F.W0 11 /r When the Source Operand is Memory and the Destination is an XMM Register)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[MAXVL-1:32] := 0
```

VMOVSS (EVEX.LLIG.F3.0F.W0 10 /r When the Source Operand is an XMM Register and the Destination is Memory)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC[31:0]
    ELSE    *DEST[31:0] remains unchanged*          ; merging-masking
FI;
```

VMOVSS (EVEX.LLIG.F3.0F.W0 10/11 /r Where the Source and Destination are XMM Registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MOVSS (Legacy SSE Version When the Source and Destination Operands are Both XMM Registers)
```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

VMOVSS (VEX.128.F3.0F 11 /r Where the Destination is an XMM Register)
```
DEST[31:0] := SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMOVSS (VEX.128.F3.0F 10 /r Where the Source and Destination are XMM Registers)
```
DEST[31:0] := SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMOVSS (VEX.128.F3.0F 10 /r When the Source Operand is Memory and the Destination is an XMM Register)
```
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0
```

MOVSS/VMOVSS (When the Source Operand is an XMM Register and the Destination is Memory)
```
DEST[31:0] := SRC[31:0]
```

MOVSS (Legacy SSE Version when the Source Operand is Memory and the Destination is an XMM Register)
```
DEST[31:0] := SRC[31:0]
DEST[127:32] := 0
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSS __m128 _mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 _mm_maskz_load_ss( __mmask8 k, float * p);
VMOVSS __m128 _mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 _mm_maskz_move_ss( __mmask8 k, __m128 s, __m128 a);
VMOVSS void _mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 _mm_load_ss(float * p)
MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-60, "Type E10 Class Exception Conditions."

MOVSX/MOVSXD—Move With Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F BE /r	MOVSX r16, r/m8[1]	RM	Valid	Valid	Move byte to word with sign-extension.
0F BE /r	MOVSX r32, r/m8[1]	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + 0F BE /r	MOVSX r64, r/m8[1]	RM	Valid	N.E.	Move byte to quadword with sign-extension.
0F BF /r	MOVSX r32, r/m16	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + 0F BF /r	MOVSX r64, r/m16	RM	Valid	N.E.	Move word to quadword with sign-extension.
63 /r[2]	MOVSXD r16, r/m16	RM	Valid	N.E.	Move word to word with sign-extension.
63 /r[1]	MOVSXD r32, r/m32	RM	Valid	N.E.	Move doubleword to doubleword with sign-extension.
REX.W + 63 /r	MOVSXD r64, r/m32	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
2. The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := SignExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MOVUPD—Move Unaligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed double precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed double precision floating-point from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed double precision floating-point from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed double precision floating-point from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed double precision floating-point from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed double precision floating-point from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 10 /r VMOVUPD xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from xmm2/m128 to xmm1 using writemask k1.
EVEX.128.66.0F.W1 11 /r VMOVUPD xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 10 /r VMOVUPD ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from ymm2/m256 to ymm1 using writemask k1.
EVEX.256.66.0F.W1 11 /r VMOVUPD ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed double precision floating-point from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPD (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPD (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*       ; merging-masking

    FI;
ENDFOR;
```

VMOVUPD (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPD (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVUPD (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVUPD (VEX.128 Encoded Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

MOVUPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPD (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVUPD __m512d _mm512_loadu_pd( void * s);
VMOVUPD __m512d _mm512_mask_loadu_pd(__m512d a, __mmask8 k, void * s);
VMOVUPD __m512d _mm512_maskz_loadu_pd( __mmask8 k, void * s);
VMOVUPD void _mm512_storeu_pd( void * d, __m512d a);
VMOVUPD void _mm512_mask_storeu_pd( void * d, __mmask8 k, __m512d a);
VMOVUPD __m256d _mm256_mask_loadu_pd(__m256d s, __mmask8 k, void * m);
VMOVUPD __m256d _mm256_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm256_mask_storeu_pd( void * d, __mmask8 k, __m256d a);
VMOVUPD __m128d _mm_mask_loadu_pd(__m128d s, __mmask8 k, void * m);
VMOVUPD __m128d _mm_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm_mask_storeu_pd( void * d, __mmask8 k, __m128d a);
MOVUPD __m256d _mm256_loadu_pd (double * p);
MOVUPD void _mm256_storeu_pd( double *p, __m256d a);
MOVUPD __m128d _mm_loadu_pd (double * p);
MOVUPD void _mm_storeu_pd( double *p, __m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Note treatment of #AC varies; additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

MOVUPS—Move Unaligned Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 10 /r MOVUPS xmm1, xmm2/m128	A	V/V	SSE	Move unaligned packed single precision floating-point from xmm2/mem to xmm1.
NP 0F 11 /r MOVUPS xmm2/m128, xmm1	B	V/V	SSE	Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.
VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed single precision floating-point from xmm2/mem to xmm1.
VEX.128.0F.WIG 11 /r VMOVUPS xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.
VEX.256.0F.WIG 10 /r VMOVUPS ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed single precision floating-point from ymm2/mem to ymm1.
VEX.256.0F.WIG 11 /r VMOVUPS ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed single precision floating-point from ymm1 to ymm2/mem.
EVEX.128.0F.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move unaligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move unaligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move unaligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPS (EVEX Encoded Versions, Register-Copy Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPS (EVEX Encoded Versions, Store-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVUPS (EVEX Encoded Versions, Load-Form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMOVUPS (VEX.256 Encoded Version, Load - and Register Copy)

```
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0
```

VMOVUPS (VEX.256 Encoded Version, Store-Form)

```
DEST[255:0] := SRC[255:0]
```

VMOVUPS (VEX.128 Encoded Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0
```

MOVUPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)

```
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPS (128-bit Store-Form Version)

```
DEST[127:0] := SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVUPS __m512 _mm512_loadu_ps( void * s);
VMOVUPS __m512 _mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 _mm512_maskz_loadu_ps( __mmask16 k, void * s);
VMOVUPS void _mm512_storeu_ps( void * d, __m512 a);
VMOVUPS void _mm512_mask_storeu_ps( void * d, __mmask8 k, __m512 a);
VMOVUPS __m256 _mm256_mask_loadu_ps(__m256 a, __mmask8 k, void * s);
VMOVUPS __m256 _mm256_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm256_mask_storeu_ps( void * d, __mmask8 k, __m256 a);
VMOVUPS __m128 _mm_mask_loadu_ps(__m128 a, __mmask8 k, void * s);
VMOVUPS __m128 _mm_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm_mask_storeu_ps( void * d, __mmask8 k, __m128 a);
MOVUPS __m256 _mm256_loadu_ps ( float * p);
MOVUPS void _mm256 _storeu_ps( float *p, __m256 a);
MOVUPS __m128 _mm_loadu_ps ( float * p);
MOVUPS void _mm_storeu_ps( float *p, __m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Note treatment of #AC varies.

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVZX—Move With Zero-Extend

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B6 /r	MOVZX r16, r/m8[1]	RM	Valid	Valid	Move byte to word with zero-extension.
0F B6 /r	MOVZX r32, r/m8[1]	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + 0F B6 /r	MOVZX r64, r/m8[1]	RM	Valid	N.E.	Move byte to quadword, zero-extension.
0F B7 /r	MOVZX r32, r/m16	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + 0F B7 /r	MOVZX r64, r/m16	RM	Valid	N.E.	Move word to quadword, zero-extension.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := ZeroExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL r/m8[1]	M	Valid	Valid	Unsigned multiply (AX := AL ∗ r/m8).
F7 /4	MUL r/m16	M	Valid	Valid	Unsigned multiply (DX:AX := AX ∗ r/m16).
F7 /4	MUL r/m32	M	Valid	Valid	Unsigned multiply (EDX:EAX := EAX ∗ r/m32).
REX.W + F7 /4	MUL r/m64	M	Valid	N.E.	Unsigned multiply (RDX:RAX := RAX ∗ r/m64).

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 4-9.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Table 4-9. MUL Results

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX
Quadword	RAX	r/m64	RDX:RAX

Operation

```
IF (Byte operation)
    THEN
        AX := AL ∗ SRC;
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                DX:AX := AX ∗ SRC;
            ELSE IF OperandSize = 32
                THEN EDX:EAX := EAX ∗ SRC; FI;
            ELSE (* OperandSize = 64 *)
                RDX:RAX := RAX ∗ SRC;
        FI;
```

```
FI;
```

Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MULPD—Multiply Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed double precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed double precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed double precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 59 /r VMULPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.
EVEX.256.66.0F.W1 59 /r VMULPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.
EVEX.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiply packed double precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] * SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] * SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMULPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64] * SRC2[127:64]
DEST[191:128] := SRC1[191:128] * SRC2[191:128]
DEST[255:192] := SRC1[255:192] * SRC2[255:192]
DEST[MAXVL-1:256] := 0;
.
```

VMULPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64] * SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

MULPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[127:64] := DEST[127:64] * SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULPD __m512d _mm512_mul_pd( __m512d a, __m512d b);
VMULPD __m512d _mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_maskz_mul_pd( __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_mul_round_pd( __m512d a, __m512d b, int);
VMULPD __m512d _mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d _mm512_maskz_mul_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d b);
MULPD __m128d _mm_mul_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MULPS—Multiply Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 59 /r MULPS xmm1, xmm2/m128	A	V/V	SSE	Multiply packed single precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed single precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed single precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.0F.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.256.0F.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiply the packed single precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPS (EVEX Encoded Version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
               DEST[i+31:i] := SRC1[i+31:i] * SRC2[31:0]
            ELSE
               DEST[i+31:i] := SRC1[i+31:i] * SRC2[i+31:i]
         FI;
      ELSE
         IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                 ; zeroing-masking
               DEST[i+31:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VMULPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[159:128] := SRC1[159:128] * SRC2[159:128]
DEST[191:160] := SRC1[191:160] * SRC2[191:160]
DEST[223:192] := SRC1[223:192] * SRC2[223:192]
DEST[255:224] := SRC1[255:224] * SRC2[255:224].
DEST[MAXVL-1:256] := 0;
```

VMULPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

MULPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[63:32] := SRC1[63:32] * SRC2[63:32]
DEST[95:64] := SRC1[95:64] * SRC2[95:64]
DEST[127:96] := SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m512 _mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 b);
MULPS __m128 _mm_mul_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

MULSD—Multiply Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	A	V/V	SSE2	Multiply the low double precision floating-point value in xmm2/m64 by low double precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.
EVEX.LLIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	C	V/V	AVX512F OR AVX10.1	Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the low double precision floating-point value in the second source operand by the low double precision floating-point value in the first source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] * SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
            FI
    FI;
ENDFOR
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VMULSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

MULSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

MULSS—Multiply Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	A	V/V	SSE	Multiply the low single precision floating-point value in xmm2/m32 by the low single precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.
EVEX.LLIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	C	V/V	AVX512F OR AVX10.1	Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the low single precision floating-point value from the second source operand by the low single precision floating-point value in the first source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the write-mask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] * SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
            FI
    FI;
ENDFOR
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VMULSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

MULSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] * SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

NEG—Two’s Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /3	NEG r/m8[1]	M	Valid	Valid	Two’s complement negate r/m8.
F7 /3	NEG r/m16	M	Valid	Valid	Two’s complement negate r/m16.
F7 /3	NEG r/m32	M	Valid	Valid	Two’s complement negate r/m32.
REX.W + F7 /3	NEG r/m64	M	Valid	N.E.	Two’s complement negate r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF DEST = 0
    THEN CF := 0;
    ELSE CF := 1;
FI;
DEST := [– (DEST)]
```

Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

NOT—One's Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /2	NOT r/m8[1]	M	Valid	Valid	Reverse each bit of r/m8.
F7 /2	NOT r/m16	M	Valid	Valid	Reverse each bit of r/m16.
F7 /2	NOT r/m32	M	Valid	Valid	Reverse each bit of r/m32.
REX.W + F7 /2	NOT r/m64	M	Valid	N.E.	Reverse each bit of r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := NOT DEST;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

OR—Logical Inclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0C ib	OR AL, imm8	I	Valid	Valid	AL OR imm8.
0D iw	OR AX, imm16	I	Valid	Valid	AX OR imm16.
0D id	OR EAX, imm32	I	Valid	Valid	EAX OR imm32.
REX.W + 0D id	OR RAX, imm32	I	Valid	N.E.	RAX OR imm32 (sign-extended).
80 /1 ib	OR r/m8[1], imm8	MI	Valid	Valid	r/m8 OR imm8.
81 /1 iw	OR r/m16, imm16	MI	Valid	Valid	r/m16 OR imm16.
81 /1 id	OR r/m32, imm32	MI	Valid	Valid	r/m32 OR imm32.
REX.W + 81 /1 id	OR r/m64, imm32	MI	Valid	N.E.	r/m64 OR imm32 (sign-extended).
83 /1 ib	OR r/m16, imm8	MI	Valid	Valid	r/m16 OR imm8 (sign-extended).
83 /1 ib	OR r/m32, imm8	MI	Valid	Valid	r/m32 OR imm8 (sign-extended).
REX.W + 83 /1 ib	OR r/m64, imm8	MI	Valid	N.E.	r/m64 OR imm8 (sign-extended).
08 /r	OR r/m8[1], r8[1]	MR	Valid	Valid	r/m8 OR r8.
09 /r	OR r/m16, r16	MR	Valid	Valid	r/m16 OR r16.
09 /r	OR r/m32, r32	MR	Valid	Valid	r/m32 OR r32.
REX.W + 09 /r	OR r/m64, r64	MR	Valid	N.E.	r/m64 OR r64.
0A /r	OR r8[1], r/m8[1]	RM	Valid	Valid	r8 OR r/m8.
0B /r	OR r16, r/m16	RM	Valid	Valid	r16 OR r/m16.
0B /r	OR r32, r/m32	RM	Valid	Valid	r32 OR r/m32.
REX.W + 0B /r	OR r64, r/m64	RM	Valid	N.E.	r64 OR r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST OR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical OR of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 56 /r VORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 56 /r VORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 56 /r VORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical OR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical OR of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VORPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE OR SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VORPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VORPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

ORPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE OR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VORPD __m512d _mm512_or_pd ( __m512d a, __m512d  b);
VORPD __m512d _mm512_mask_or_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d _mm512_maskz_or_pd (__mmask8 k, __m512d a, __m512d b);
VORPD __m256d _mm256_mask_or_pd (__m256d s, ___mmask8 k, __m256d a, __m256d b);
VORPD __m256d _mm256_maskz_or_pd (__mmask8 k, __m256d a, __m256d b);
VORPD __m128d _mm_mask_or_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d _mm_maskz_or_pd (__mmask8 k, __m128d a, __m128d b);
VORPD __m256d _mm256_or_pd (__m256d a, __m256d b);
ORPD __m128d _mm_or_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 56 /r ORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical OR of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical OR of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical OR of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VORPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VORPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE OR SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VORPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

ORPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 _mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 _mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 _mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 _mm256_mask_or_ps (__m256 s, ___mmask8 k, __m256 a, __m256 b);
VORPS __m256 _mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 _mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 _mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 1C /r[1] PABSB mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1C /r PABSB xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
NP 0F 38 1D /r[1] PABSW mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
NP 0F 38 1E /r[1] PABSD mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1C /r VPABSB xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1D /r VPABSW xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 16- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1E /r VPABSD xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 32- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.256.66.0F38.WIG 1C /r VPABSB ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1D /r VPABSW ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1E /r VPABSD ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
EVEX.128.66.0F38.WIG 1C /r VPABSB xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1C /r VPABSB ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1C /r VPABSB zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW OR AVX10.1	Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.WIG 1D /r VPABSW xmm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW OR AVX10.1	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1E /r VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded versions: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PABSB With 64-bit Operands:

```
    Unsigned DEST[7:0] := ABS(SRC[7: 0])
    Repeat operation for 2nd through 7th bytes
    Unsigned DEST[63:56] := ABS(SRC[63:56])
```

PABSB With 128-bit Operands:

```
    Unsigned DEST[7:0] := ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120] := ABS(SRC[127:120])
```

VPABSB With 128-bit Operands:

```
    Unsigned DEST[7:0] := ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120] := ABS(SRC[127:120])
```

VPABSB With 256-bit Operands:

```
    Unsigned DEST[7:0] := ABS(SRC[7: 0])
    Repeat operation for 2nd through 31st bytes
    Unsigned DEST[255:248] := ABS(SRC[255:248])
```

VPABSB (EVEX Encoded Versions)

```
    (KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+7:i] := ABS(SRC[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PABSW With 128-bit Operands:

```
    Unsigned DEST[15:0] := ABS(SRC[15:0])
    Repeat operation for 2nd through 7th 16-bit words
    Unsigned DEST[127:112] := ABS(SRC[127:112])
```

VPABSW With 128-bit Operands:

```
    Unsigned DEST[15:0] := ABS(SRC[15:0])
    Repeat operation for 2nd through 7th 16-bit words
    Unsigned DEST[127:112] := ABS(SRC[127:112])
```

VPABSW With 256-bit Operands:

```
    Unsigned DEST[15:0] := ABS(SRC[15:0])
    Repeat operation for 2nd through 15th 16-bit words
    Unsigned DEST[255:240] := ABS(SRC[255:240])
```

VPABSW (EVEX Encoded Versions)

```
    (KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+15:i] := ABS(SRC[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PABSD With 128-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] := ABS(SRC[127:96])
```

VPABSD With 128-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] := ABS(SRC[127:96])
```

VPABSD With 256-bit Operands:

```
    Unsigned DEST[31:0] := ABS(SRC[31:0])
    Repeat operation for 2nd through 7th 32-bit double words
    Unsigned DEST[255:224] := ABS(SRC[255:224])
```

VPABSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+31:i] := ABS(SRC[31:0])
                ELSE
                    Unsigned DEST[i+31:i] := ABS(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
```

```
DEST[MAXVL-1:VL] := 0
```

VPABSQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN
               Unsigned DEST[i+63:i] := ABS(SRC[63:0])
            ELSE
               Unsigned DEST[i+63:i] := ABS(SRC[i+63:i])
         FI;
      ELSE
         IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking*               ; zeroing-masking
               DEST[i+63:i] := 0
         FI
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPABSB__m512i _mm512_abs_epi8 ( __m512i a)
VPABSW__m512i _mm512_abs_epi16 ( __m512i a)
VPABSB__m512i _mm512_mask_abs_epi8 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i _mm512_mask_abs_epi16 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i _mm512_maskz_abs_epi8 (__mmask64 m, __m512i a)
VPABSW__m512i _mm512_maskz_abs_epi16 (__mmask32 m, __m512i a)
VPABSB__m256i _mm256_mask_abs_epi8 (__m256i s, __mmask32 m, __m256i a)
VPABSW__m256i _mm256_mask_abs_epi16 (__m256i s, __mmask16 m, __m256i a)
VPABSB__m256i _mm256_maskz_abs_epi8 (__mmask32 m, __m256i a)
VPABSW__m256i _mm256_maskz_abs_epi16 (__mmask16 m, __m256i a)
VPABSB__m128i _mm_mask_abs_epi8 (__m128i s, __mmask16 m, __m128i a)
VPABSW__m128i _mm_mask_abs_epi16 (__m128i s, __mmask8 m, __m128i a)
VPABSB__m128i _mm_maskz_abs_epi8 (__mmask16 m, __m128i a)
VPABSW__m128i _mm_maskz_abs_epi16 (__mmask8 m, __m128i a)
VPABSD __m256i _mm256_mask_abs_epi32(__m256i s, __mmask8 k, __m256i a);
VPABSD __m256i _mm256_maskz_abs_epi32( __mmask8 k, __m256i a);
VPABSD __m128i _mm_mask_abs_epi32(__m128i s, __mmask8 k, __m128i a);
VPABSD __m128i _mm_maskz_abs_epi32( __mmask8 k, __m128i a);
VPABSD  __m512i _mm512_abs_epi32( __m512i a);
VPABSD  __m512i _mm512_mask_abs_epi32(__m512i s, __mmask16 k, __m512i a);
VPABSD  __m512i _mm512_maskz_abs_epi32( __mmask16 k, __m512i a);
VPABSQ __m512i _mm512_abs_epi64( __m512i a);
VPABSQ __m512i _mm512_mask_abs_epi64(__m512i s, __mmask8 k, __m512i a);
VPABSQ __m512i _mm512_maskz_abs_epi64( __mmask8 k, __m512i a);
VPABSQ __m256i _mm256_mask_abs_epi64(__m256i s, __mmask8 k, __m256i a);
VPABSQ __m256i _mm256_maskz_abs_epi64( __mmask8 k, __m256i a);
VPABSQ __m128i _mm_mask_abs_epi64(__m128i s, __mmask8 k, __m128i a);
VPABSQ __m128i _mm_maskz_abs_epi64( __mmask8 k, __m128i a);
```

```
PABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m256i _mm256_abs_epi8 (__m256i a)
PABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m256i _mm256_abs_epi16 (__m256i a)
PABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m256i _mm256_abs_epi32 (__m256i a)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPABSD/Q, see Table 2-51, “Type E4 Class Exception Conditions.”

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PACKSSWB/PACKSSDW—Pack With Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 63 /r[1] PACKSSWB mm1, mm2/m64	A	V/V	MMX	Converts 4 packed signed word integers from mm1 and from mm2/m64 into 8 packed signed byte integers in mm1 using signed saturation.
66 0F 63 /r PACKSSWB xmm1, xmm2/m128	A	V/V	SSE2	Converts 8 packed signed word integers from xmm1 and from xmm2/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
NP 0F 6B /r[1] PACKSSDW mm1, mm2/m64	A	V/V	MMX	Converts 2 packed signed doubleword integers from mm1 and from mm2/m64 into 4 packed signed word integers in mm1 using signed saturation.
66 0F 6B /r PACKSSDW xmm1, xmm2/m128	A	V/V	SSE2	Converts 4 packed signed doubleword integers from xmm1 and from xmm2/m128 into 8 packed signed word integers in xmm1 using signed saturation.
VEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 8 packed signed word integers from xmm2 and from xmm3/m128 into 16 packed signed byte integers in xmm1 using signed saturation.
VEX.128.66.0F.WIG 6B /r VPACKSSDW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Converts 4 packed signed doubleword integers from xmm2 and from xmm3/m128 into 8 packed signed word integers in xmm1 using signed saturation.
VEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 16 packed signed word integers from ymm2 and from ymm3/m256 into 32 packed signed byte integers in ymm1 using signed saturation.
VEX.256.66.0F.WIG 6B /r VPACKSSDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 8 packed signed doubleword integers from ymm2 and from ymm3/m256 into 16 packed signed word integers in ymm1using signed saturation.
EVEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed word integers from xmm2 and from xmm3/m128 into packed signed byte integers in xmm1 using signed saturation under writemask k1.
EVEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed word integers from ymm2 and from ymm3/m256 into packed signed byte integers in ymm1 using signed saturation under writemask k1.
EVEX.512.66.0F.WIG 63 /r VPACKSSWB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Converts packed signed word integers from zmm2 and from zmm3/m512 into packed signed byte integers in zmm1 using signed saturation under writemask k1.
EVEX.128.66.0F.W0 6B /r VPACKSSDW xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed doubleword integers from xmm2 and from xmm3/m128/m32bcst into packed signed word integers in xmm1 using signed saturation under writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts packed signed doubleword integers from ymm2 and from ymm3/m256/m32bcst into packed signed word integers in ymm1 using signed saturation under writemask k1.
EVEX.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512BW OR AVX10.1	Converts packed signed doubleword integers from zmm2 and from zmm3/m512/m32bcst into packed signed word integers in zmm1 using signed saturation under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.

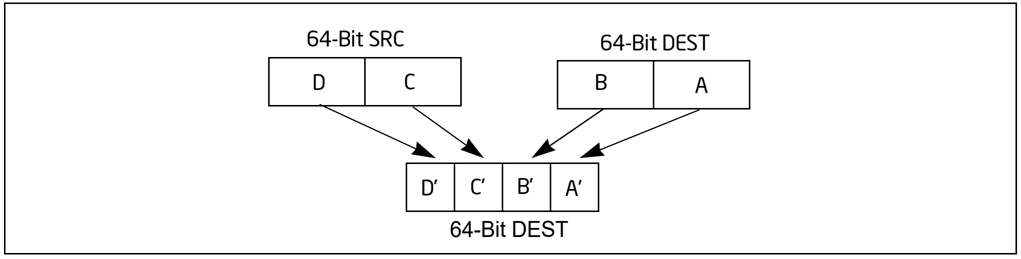


Figure 4-6. Operation of the PACKSSDW Instruction Using 64-Bit Operands

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed word value is beyond the range of a signed byte value (i.e., greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-

bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM destination register destination are unmodified.

Operation

PACKSSWB Instruction (128-bit Legacy SSE Version)

```
    DEST[7:0] := SaturateSignedWordToSignedByte (DEST[15:0]);
    DEST[15:8] := SaturateSignedWordToSignedByte (DEST[31:16]);
    DEST[23:16] := SaturateSignedWordToSignedByte (DEST[47:32]);
    DEST[31:24] := SaturateSignedWordToSignedByte (DEST[63:48]);
    DEST[39:32] := SaturateSignedWordToSignedByte (DEST[79:64]);
    DEST[47:40] := SaturateSignedWordToSignedByte (DEST[95:80]);
    DEST[55:48] := SaturateSignedWordToSignedByte (DEST[111:96]);
    DEST[63:56] := SaturateSignedWordToSignedByte (DEST[127:112]);
    DEST[71:64] := SaturateSignedWordToSignedByte (SRC[15:0]);
    DEST[79:72] := SaturateSignedWordToSignedByte (SRC[31:16]);
    DEST[87:80] := SaturateSignedWordToSignedByte (SRC[47:32]);
    DEST[95:88] := SaturateSignedWordToSignedByte (SRC[63:48]);
    DEST[103:96] := SaturateSignedWordToSignedByte (SRC[79:64]);
    DEST[111:104] := SaturateSignedWordToSignedByte (SRC[95:80]);
    DEST[119:112] := SaturateSignedWordToSignedByte (SRC[111:96]);
    DEST[127:120] := SaturateSignedWordToSignedByte (SRC[127:112]);
    DEST[MAXVL-1:128] (Unmodified)
```

PACKSSDW Instruction (128-bit Legacy SSE Version)

```
    DEST[15:0] := SaturateSignedDwordToSignedWord (DEST[31:0]);
    DEST[31:16] := SaturateSignedDwordToSignedWord (DEST[63:32]);
    DEST[47:32] := SaturateSignedDwordToSignedWord (DEST[95:64]);
    DEST[63:48] := SaturateSignedDwordToSignedWord (DEST[127:96]);
    DEST[79:64] := SaturateSignedDwordToSignedWord (SRC[31:0]);
    DEST[95:80] := SaturateSignedDwordToSignedWord (SRC[63:32]);
    DEST[111:96] := SaturateSignedDwordToSignedWord (SRC[95:64]);
    DEST[127:112] := SaturateSignedDwordToSignedWord (SRC[127:96]);
    DEST[MAXVL-1:128] (Unmodified)
```

VPACKSSWB Instruction (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] := 0;
```

VPACKSSDW Instruction (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] := SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[MAXVL-1:128] := 0;
```

VPACKSSWB Instruction (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[135:128] := SaturateSignedWordToSignedByte (SRC1[143:128]);
DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
DEST[151:144] := SaturateSignedWordToSignedByte (SRC1[175:160]);
DEST[159:152] := SaturateSignedWordToSignedByte (SRC1[191:176]);
DEST[167:160] := SaturateSignedWordToSignedByte (SRC1[207:192]);
DEST[175:168] := SaturateSignedWordToSignedByte (SRC1[223:208]);
DEST[183:176] := SaturateSignedWordToSignedByte (SRC1[239:224]);
```

```
    DEST[191:184] := SaturateSignedWordToSignedByte (SRC1[255:240]);
    DEST[199:192] := SaturateSignedWordToSignedByte (SRC2[143:128]);
    DEST[207:200] := SaturateSignedWordToSignedByte (SRC2[159:144]);
    DEST[215:208] := SaturateSignedWordToSignedByte (SRC2[175:160]);
    DEST[223:216] := SaturateSignedWordToSignedByte (SRC2[191:176]);
    DEST[231:224] := SaturateSignedWordToSignedByte (SRC2[207:192]);
    DEST[239:232] := SaturateSignedWordToSignedByte (SRC2[223:208]);
    DEST[247:240] := SaturateSignedWordToSignedByte (SRC2[239:224]);
    DEST[255:248] := SaturateSignedWordToSignedByte (SRC2[255:240]);
    DEST[MAXVL-1:256] := 0;
```

VPACKSSDW Instruction (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
    DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
    DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
    DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
    DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
    DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
    DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
    DEST[127:112] := SaturateSignedDwordToSignedWord (SRC2[127:96]);
    DEST[143:128] := SaturateSignedDwordToSignedWord (SRC1[159:128]);
    DEST[159:144] := SaturateSignedDwordToSignedWord (SRC1[191:160]);
    DEST[175:160] := SaturateSignedDwordToSignedWord (SRC1[223:192]);
    DEST[191:176] := SaturateSignedDwordToSignedWord (SRC1[255:224]);
    DEST[207:192] := SaturateSignedDwordToSignedWord (SRC2[159:128]);
    DEST[223:208] := SaturateSignedDwordToSignedWord (SRC2[191:160]);
    DEST[239:224] := SaturateSignedDwordToSignedWord (SRC2[223:192]);
    DEST[255:240] := SaturateSignedDwordToSignedWord (SRC2[255:224]);
    DEST[MAXVL-1:256] := 0;
```

VPACKSSWB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[15:8] := SaturateSignedWordToSignedByte (SRC1[31:16]);
TMP_DEST[23:16] := SaturateSignedWordToSignedByte (SRC1[47:32]);
TMP_DEST[31:24] := SaturateSignedWordToSignedByte (SRC1[63:48]);
TMP_DEST[39:32] := SaturateSignedWordToSignedByte (SRC1[79:64]);
TMP_DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[55:48] := SaturateSignedWordToSignedByte (SRC1[111:96]);
TMP_DEST[63:56] := SaturateSignedWordToSignedByte (SRC1[127:112]);
TMP_DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[79:72] := SaturateSignedWordToSignedByte (SRC2[31:16]);
TMP_DEST[87:80] := SaturateSignedWordToSignedByte (SRC2[47:32]);
TMP_DEST[95:88] := SaturateSignedWordToSignedByte (SRC2[63:48]);
TMP_DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] := SaturateSignedWordToSignedByte (SRC2[111:96]);
TMP_DEST[127:120] := SaturateSignedWordToSignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] := SaturateSignedWordToSignedByte (SRC1[143:128]);
    TMP_DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
    TMP_DEST[151:144] := SaturateSignedWordToSignedByte (SRC1[175:160]);
    TMP_DEST[159:152] := SaturateSignedWordToSignedByte (SRC1[191:176]);
    TMP_DEST[167:160] := SaturateSignedWordToSignedByte (SRC1[207:192]);
    TMP_DEST[175:168] := SaturateSignedWordToSignedByte (SRC1[223:208]);
    TMP_DEST[183:176] := SaturateSignedWordToSignedByte (SRC1[239:224]);
    TMP_DEST[191:184] := SaturateSignedWordToSignedByte (SRC1[255:240]);
    TMP_DEST[199:192] := SaturateSignedWordToSignedByte (SRC2[143:128]);
    TMP_DEST[207:200] := SaturateSignedWordToSignedByte (SRC2[159:144]);
    TMP_DEST[215:208] := SaturateSignedWordToSignedByte (SRC2[175:160]);
    TMP_DEST[223:216] := SaturateSignedWordToSignedByte (SRC2[191:176]);
    TMP_DEST[231:224] := SaturateSignedWordToSignedByte (SRC2[207:192]);
    TMP_DEST[239:232] := SaturateSignedWordToSignedByte (SRC2[223:208]);
    TMP_DEST[247:240] := SaturateSignedWordToSignedByte (SRC2[239:224]);
    TMP_DEST[255:248] := SaturateSignedWordToSignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] := SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] := SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] := SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] := SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] := SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] := SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] := SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] := SaturateSignedWordToSignedByte (SRC1[383:368]);

    TMP_DEST[327:320] := SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] := SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] := SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] := SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] := SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] := SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] := SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] := SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[391:384] := SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] := SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] := SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] := SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] := SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] := SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] := SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] := SaturateSignedWordToSignedByte (SRC1[511:496]);

    TMP_DEST[455:448] := SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] := SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] := SaturateSignedWordToSignedByte (SRC2[431:416]);
    TMP_DEST[479:472] := SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] := SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] := SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] := SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] := SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPACKSSDW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO ((KL/2) - 1)
    i := j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;

TMP_DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] := SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] := SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] := SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] := SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] := SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
    TMP_DEST[143:128] := SaturateSignedDwordToSignedWord (SRC1[159:128]);
    TMP_DEST[159:144] := SaturateSignedDwordToSignedWord (SRC1[191:160]);
    TMP_DEST[175:160] := SaturateSignedDwordToSignedWord (SRC1[223:192]);
    TMP_DEST[191:176] := SaturateSignedDwordToSignedWord (SRC1[255:224]);
    TMP_DEST[207:192] := SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
    TMP_DEST[223:208] := SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
    TMP_DEST[239:224] := SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
    TMP_DEST[255:240] := SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
    TMP_DEST[271:256] := SaturateSignedDwordToSignedWord (SRC1[287:256]);
    TMP_DEST[287:272] := SaturateSignedDwordToSignedWord (SRC1[319:288]);
    TMP_DEST[303:288] := SaturateSignedDwordToSignedWord (SRC1[351:320]);
    TMP_DEST[319:304] := SaturateSignedDwordToSignedWord (SRC1[383:352]);
    TMP_DEST[335:320] := SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
    TMP_DEST[351:336] := SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
    TMP_DEST[367:352] := SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
    TMP_DEST[383:368] := SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

    TMP_DEST[399:384] := SaturateSignedDwordToSignedWord (SRC1[415:384]);
    TMP_DEST[415:400] := SaturateSignedDwordToSignedWord (SRC1[447:416]);
    TMP_DEST[431:416] := SaturateSignedDwordToSignedWord (SRC1[479:448]);
    TMP_DEST[447:432] := SaturateSignedDwordToSignedWord (SRC1[511:480]);
    TMP_DEST[463:448] := SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
    TMP_DEST[479:464] := SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
    TMP_DEST[495:480] := SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
    TMP_DEST[511:496] := SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPACKSSDW__m512i _mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_maskz_packs_epi32( __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m256i _mm256_mask_packs_epi32( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_maskz_packs_epi32( __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m128i _mm_mask_packs_epi32( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW__m128i _mm_maskz_packs_epi32( __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m512i _mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_maskz_packs_epi16( __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m256i _mm256_mask_packs_epi16( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m256i _mm256_maskz_packs_epi16( __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m128i _mm_mask_packs_epi16( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m128i _mm_maskz_packs_epi16( __mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPACKSSDW, see Table 2-52, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PACKUSDW—Pack With Unsigned Saturation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW xmm1, xmm2/m128	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from xmm1 and 4 packed signed doubleword integers from xmm2/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.
VEX.128.66.0F38 2B /r VPACKUSDW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Convert 4 packed signed doubleword integers from xmm2 and 4 packed signed doubleword integers from xmm3/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.
VEX.256.66.0F38 2B /r VPACKUSDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Convert 8 packed signed doubleword integers from ymm2 and 8 packed signed doubleword integers from ymm3/m256 into 16 packed unsigned word integers in ymm1 using unsigned saturation.
EVEX.128.66.0F38.W0 2B /r VPACKUSDW xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Convert packed signed doubleword integers from xmm2 and packed signed doubleword integers from xmm3/m128/m32bcst into packed unsigned word integers in xmm1 using unsigned saturation under writemask k1.
EVEX.256.66.0F38.W0 2B /r VPACKUSDW ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Convert packed signed doubleword integers from ymm2 and packed signed doubleword integers from ymm3/m256/m32bcst into packed unsigned word integers in ymm1 using unsigned saturation under writemask k1.
EVEX.512.66.0F38.W0 2B /r VPACKUSDW zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512BW OR AVX10.1	Convert packed signed doubleword integers from zmm2 and packed signed doubleword integers from zmm3/m512/m32bcst into packed unsigned word integers in zmm1 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding destination register destination are unmodified.

Operation

PACKUSDW (Legacy SSE Instruction)

```
TMP[15:0] := (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] := (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] := (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] := (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] := (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[79:64] := (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] := (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] := (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[127:112] := (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
DEST[MAXVL-1:128] (Unmodified)
```

PACKUSDW (VEX.128 Encoded Version)

```
TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] := (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] := (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] := (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] := (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[MAXVL-1:128] := 0;
```

VPACKUSDW (VEX.256 Encoded Version)

```
TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] := (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] := (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] := (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] := (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
TMP[143:128] := (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] := (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
TMP[159:144] := (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] := (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
TMP[175:160] := (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] := (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
TMP[191:176] := (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] := (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
TMP[207:192] := (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] := (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
TMP[223:208] := (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] := (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
TMP[239:224] := (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] := (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
TMP[255:240] := (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] := (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
DEST[MAXVL-1:256] := 0;
```

VPACKUSDW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO ((KL/2) - 1)
    i := j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;

TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] := (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] := (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] := (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
```

```
TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] := (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] := (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] := (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] := (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] := (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] := (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] := (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] := (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] := (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
IF VL >= 256
    TMP[143:128] := (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
    DEST[143:128] := (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
    TMP[159:144] := (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
    DEST[159:144] := (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
    TMP[175:160] := (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
    DEST[175:160] := (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
    TMP[191:176] := (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
    DEST[191:176] := (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
    TMP[207:192] := (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
    DEST[207:192] := (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
    TMP[223:208] := (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
    DEST[223:208] := (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
    TMP[239:224] := (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
    DEST[239:224] := (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
    TMP[255:240] := (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
    DEST[255:240] := (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
FI;
IF VL >= 512
    TMP[271:256] := (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
    DEST[271:256] := (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256] ;
    TMP[287:272] := (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
    DEST[287:272] := (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272] ;
    TMP[303:288] := (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
    DEST[303:288] := (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288] ;
    TMP[319:304] := (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
    DEST[319:304] := (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304] ;
    TMP[335:320] := (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
    DEST[335:304] := (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64] ;
    TMP[351:336] := (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
    DEST[351:336] := (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336] ;
    TMP[367:352] := (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
    DEST[367:352] := (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352] ;
    TMP[383:368] := (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
    DEST[383:368] := (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368] ;
    TMP[399:384] := (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
    DEST[399:384] := (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384] ;
    TMP[415:400] := (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
    DEST[415:400] := (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400] ;
    TMP[431:416] := (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
    DEST[431:416] := (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416] ;
    TMP[447:432] := (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
    DEST[447:432] := (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432] ;
    TMP[463:448] := (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
    DEST[463:448] := (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448] ;
    TMP[475:464] := (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
    DEST[475:464] := (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464] ;
    TMP[491:476] := (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
    DEST[491:476] := (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476] ;
    TMP[511:492] := (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
    DEST[511:492] := (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492] ;
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPACKUSDW__m512i _mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_maskz_packus_epi32( __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i _mm256_mask_packus_epi32( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i _mm256_maskz_packus_epi32( __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i _mm_mask_packus_epi32( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i _mm_maskz_packus_epi32( __mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i _mm256_packus_epi32(__m256i m1, __m256i m2);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

PACKUSWB—Pack With Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 67 /r[1] PACKUSWB mm, mm/m64	A	V/V	MMX	Converts 4 signed word integers from mm and 4 signed word integers from mm/m64 into 8 unsigned byte integers in mm using unsigned saturation.
66 0F 67 /r PACKUSWB xmm1, xmm2/m128	A	V/V	SSE2	Converts 8 signed word integers from xmm1 and 8 signed word integers from xmm2/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
VEX.128.66.0F.WIG 67 /r VPACKUSWB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Converts 8 signed word integers from xmm2 and 8 signed word integers from xmm3/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.
VEX.256.66.0F.WIG 67 /r VPACKUSWB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Converts 16 signed word integers from ymm2 and 16signed word integers from ymm3/m256 into 32 unsigned byte integers in ymm1 using unsigned saturation.
EVEX.128.66.0F.WIG 67 /r VPACKUSWB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts signed word integers from xmm2 and signed word integers from xmm3/m128 into unsigned byte integers in xmm1 using unsigned saturation under writemask k1.
EVEX.256.66.0F.WIG 67 /r VPACKUSWB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts signed word integers from ymm2 and signed word integers from ymm3/m256 into unsigned byte integers in ymm1 using unsigned saturation under writemask k1.
EVEX.512.66.0F.WIG 67 /r VPACKUSWB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Converts signed word integers from zmm2 and signed word integers from zmm3/m512 into unsigned byte integers in zmm1 using unsigned saturation under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts 4, 8, 16, or 32 signed word integers from the destination operand (first operand) and 4, 8, 16, or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PACKUSWB (With 64-bit Operands)

```
    DEST[7:0] := SaturateSignedWordToUnsignedByte DEST[15:0];
    DEST[15:8] := SaturateSignedWordToUnsignedByte DEST[31:16];
    DEST[23:16] := SaturateSignedWordToUnsignedByte DEST[47:32];
    DEST[31:24] := SaturateSignedWordToUnsignedByte DEST[63:48];
    DEST[39:32] := SaturateSignedWordToUnsignedByte SRC[15:0];
    DEST[47:40] := SaturateSignedWordToUnsignedByte SRC[31:16];
    DEST[55:48] := SaturateSignedWordToUnsignedByte SRC[47:32];
    DEST[63:56] := SaturateSignedWordToUnsignedByte SRC[63:48];
```

PACKUSWB (Legacy SSE Instruction)

```
    DEST[7:0] := SaturateSignedWordToUnsignedByte (DEST[15:0]);
    DEST[15:8] := SaturateSignedWordToUnsignedByte (DEST[31:16]);
    DEST[23:16] := SaturateSignedWordToUnsignedByte (DEST[47:32]);
    DEST[31:24] := SaturateSignedWordToUnsignedByte (DEST[63:48]);
    DEST[39:32] := SaturateSignedWordToUnsignedByte (DEST[79:64]);
    DEST[47:40] := SaturateSignedWordToUnsignedByte (DEST[95:80]);
    DEST[55:48] := SaturateSignedWordToUnsignedByte (DEST[111:96]);
    DEST[63:56] := SaturateSignedWordToUnsignedByte (DEST[127:112]);
    DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC[15:0]);
    DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC[31:16]);
    DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC[47:32]);
    DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC[63:48]);
    DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC[79:64]);
    DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC[95:80]);
    DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC[111:96]);
    DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

PACKUSWB (VEX.128 Encoded Version)

```
    DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
    DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
    DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
    DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
    DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
    DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
    DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
    DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
    DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
    DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
    DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
    DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
    DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
    DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
    DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[MAXVL-1:128] := 0;
```

VPACKUSWB (VEX.256 Encoded Version)

```
    DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
    DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
    DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
    DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
    DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
    DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
    DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
    DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
    DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
    DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
    DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
    DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
    DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
    DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
    DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[135:128] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    DEST[167:160] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    DEST[215:208] := SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    DEST[223:216] := SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    DEST[231:224] := SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    DEST[239:232] := SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    DEST[247:240] := SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    DEST[255:248] := SaturateSignedWordToUnsignedByte (SRC2[255:240]);
```

VPACKUSWB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
TMP_DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
TMP_DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC2[15:0]);
TMP_DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC2[31:16]);
TMP_DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC2[47:32]);
TMP_DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC2[63:48]);
TMP_DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC2[79:64]);
TMP_DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

```
TMP_DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] := SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] := SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] := SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] := SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] := SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] := SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] := SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] := SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] := SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] := SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] := SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] := SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] := SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] := SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] := SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] := SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] := SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] := SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] := SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] := SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] := SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] := SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] := SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] := SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] := SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] := SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] := SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] := SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] := SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] := SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] := SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] := SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] := SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] := SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] := SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] := SaturateSignedWordToUnsignedByte (SRC2[479:464]);
    TMP_DEST[503:496] := SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] := SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPACKUSWB __m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m256i _mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB __m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);
PACKUSWB __m64 _mm_packs_pu16(__m64 m1, __m64 m2)
(V)PACKUSWB __m128i _mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	**Op / En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F FC /r[1] PADDB mm, mm/m64	A	V/V	MMX	Add packed byte integers from mm/m64 and mm.
NP 0F FD /r[1] PADDW mm, mm/m64	A	V/V	MMX	Add packed word integers from mm/m64 and mm.
NP 0F FE /r[1] PADDD mm, mm/m64	A	V/V	MMX	Add packed doubleword integers from mm/m64 and mm.
NP 0F D4 /r[1] PADDQ mm, mm/m64	A	V/V	MMX	Add packed quadword integers from mm/m64 and mm.
66 0F FC /r PADDB xmm1, xmm2/m128	A	V/V	SSE2	Add packed byte integers from xmm2/m128 and xmm1.
66 0F FD /r PADDW xmm1, xmm2/m128	A	V/V	SSE2	Add packed word integers from xmm2/m128 and xmm1.
66 0F FE /r PADDD xmm1, xmm2/m128	A	V/V	SSE2	Add packed doubleword integers from xmm2/m128 and xmm1.
66 0F D4 /r PADDQ xmm1, xmm2/m128	A	V/V	SSE2	Add packed quadword integers from xmm2/m128 and xmm1.
VEX.128.66.0F.WIG FC /r VPADDB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG FD /r VPADDW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed word integers from xmm2, xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG FE /r VPADDD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed doubleword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.128.66.0F.WIG D4 /r VPADDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed quadword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.256.66.0F.WIG FC /r VPADDB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG FD /r VPADDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed word integers from ymm2, ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG FE /r VPADDD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed doubleword integers from ymm2, ymm3/m256 and store in ymm1.
VEX.256.66.0F.WIG D4 /r VPADDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed quadword integers from ymm2, ymm3/m256 and store in ymm1.
EVEX.128.66.0F.WIG FC /r VPADDB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.128.66.0F.WIG FD /r VPADDW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed word integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.128.66.0F.W0 FE /r VPADDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed doubleword integers from xmm2, and xmm3/m128/m32bcst and store in xmm1 using writemask k1.
EVEX.128.66.0F.W1 D4 /r VPADDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed quadword integers from xmm2, and xmm3/m128/m64bcst and store in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG FC /r VPADDB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.
EVEX.256.66.0F.WIG FD /r VPADDW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed word integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.
EVEX.256.66.0F.W0 FE /r VPADDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed doubleword integers from ymm2, ymm3/m256/m32bcst and store in ymm1 using writemask k1.
EVEX.256.66.0F.W1 D4 /r VPADDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Add packed quadword integers from ymm2, ymm3/m256/m64bcst and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG FC /r VPADDB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed byte integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.
EVEX.512.66.0F.WIG FD /r VPADDW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed word integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.
EVEX.512.66.0F.W0 FE /r VPADDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Add packed doubleword integers from zmm2, zmm3/m512/m32bcst and store in zmm1 using writemask k1.
EVEX.512.66.0F.W1 D4 /r VPADDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Add packed quadword integers from zmm2, zmm3/m512/m64bcst and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. the upper bits (MAXVL-1:256) of the destination are cleared.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

PADDB (With 64-bit Operands)

```
    DEST[7:0] := DEST[7:0] + SRC[7:0];
    (* Repeat add operation for 2nd through 7th byte *)
    DEST[63:56] := DEST[63:56] + SRC[63:56];
```

PADDW (With 64-bit Operands)

```
    DEST[15:0] := DEST[15:0] + SRC[15:0];
    (* Repeat add operation for 2nd and 3th word *)
    DEST[63:48] := DEST[63:48] + SRC[63:48];
```

PADDD (With 64-bit Operands)

```
    DEST[31:0] := DEST[31:0] + SRC[31:0];
    DEST[63:32] := DEST[63:32] + SRC[63:32];
```

PADDQ (With 64-Bit Operands)

```
    DEST[63:0] := DEST[63:0] + SRC[63:0];
```

PADDB (Legacy SSE Instruction)
```
DEST[7:0] := DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120] := DEST[127:120] + SRC[127:120];
DEST[MAXVL-1:128] (Unmodified)
```

PADDW (Legacy SSE Instruction)
```
DEST[15:0] := DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] := DEST[127:112] + SRC[127:112];
DEST[MAXVL-1:128] (Unmodified)
```

PADDD (Legacy SSE Instruction)
```
DEST[31:0] := DEST[31:0] + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] := DEST[127:96] + SRC[127:96];
DEST[MAXVL-1:128] (Unmodified)
```

PADDQ (Legacy SSE Instruction)
```
DEST[63:0] := DEST[63:0] + SRC[63:0];
DEST[127:64] := DEST[127:64] + SRC[127:64];
DEST[MAXVL-1:128] (Unmodified)
```

VPADDB (VEX.128 Encoded Instruction)
```
DEST[7:0] := SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120] := SRC1[127:120] + SRC2[127:120];
DEST[MAXVL-1:128] := 0;
```

VPADDW (VEX.128 Encoded Instruction)
```
DEST[15:0] := SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] := SRC1[127:112] + SRC2[127:112];
DEST[MAXVL-1:128] := 0;
```

VPADDD (VEX.128 Encoded Instruction)
```
DEST[31:0] := SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] := SRC1[127:96] + SRC2[127:96];
DEST[MAXVL-1:128] := 0;
```

VPADDQ (VEX.128 Encoded Instruction)
```
DEST[63:0] := SRC1[63:0] + SRC2[63:0];
DEST[127:64] := SRC1[127:64] + SRC2[127:64];
DEST[MAXVL-1:128] := 0;
```

VPADDB (VEX.256 Encoded Instruction)
```
DEST[7:0] := SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248] := SRC1[255:248] + SRC2[255:248];
```

VPADDW (VEX.256 Encoded Instruction)

```
    DEST[15:0] := SRC1[15:0] + SRC2[15:0];
    (* Repeat add operation for 2nd through 15th word *)
    DEST[255:240] := SRC1[255:240] + SRC2[255:240];
```

VPADDD (VEX.256 Encoded Instruction)

```
    DEST[31:0] := SRC1[31:0] + SRC2[31:0];
    (* Repeat add operation for 2nd and 7th doubleword *)
    DEST[255:224] := SRC1[255:224] + SRC2[255:224];
```

VPADDQ (VEX.256 Encoded Instruction)

```
    DEST[63:0] := SRC1[63:0] + SRC2[63:0];
    DEST[127:64] := SRC1[127:64] + SRC2[127:64];
    DEST[191:128] := SRC1[191:128] + SRC2[191:128];
    DEST[255:192] := SRC1[255:192] + SRC2[255:192];
```

VPADDB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC1[i+7:i] + SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC1[i+15:i] + SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] + SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] + SRC2[63:0]
                ELSE DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPADDB__m512i _mm512_add_epi8 ( __m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 ( __m512i a, __m512i b)
VPADDB__m512i _mm512_mask_add_epi8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_mask_add_epi16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m512i _mm512_maskz_add_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_maskz_add_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_maskz_add_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_maskz_add_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_mask_add_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
```

```
VPADDB__m128i _mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD __m512i _mm512_add_epi32( __m512i a, __m512i b);
VPADDD __m512i _mm512_mask_add_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPADDD __m512i _mm512_maskz_add_epi32( __mmask16 k, __m512i a, __m512i b);
VPADDD __m256i _mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD __m256i _mm256_maskz_add_epi32( __mmask8 k, __m256i a, __m256i b);
VPADDD __m128i _mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDD __m128i _mm_maskz_add_epi32( __mmask8 k, __m128i a, __m128i b);
VPADDQ __m512i _mm512_add_epi64( __m512i a, __m512i b);
VPADDQ __m512i _mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ __m512i _mm512_maskz_add_epi64( __mmask8 k, __m512i a, __m512i b);
VPADDQ __m256i _mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ __m256i _mm256_maskz_add_epi64( __mmask8 k, __m256i a, __m256i b);
VPADDQ __m128i _mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ __m128i _mm_maskz_add_epi64( __mmask8 k, __m128i a, __m128i b);
PADDB __m128i _mm_add_epi8 (__m128i a,__m128i b );
PADDW __m128i _mm_add_epi16 ( __m128i a, __m128i b);
PADDD  __m128i _mm_add_epi32 ( __m128i a, __m128i b);
PADDQ __m128i _mm_add_epi64 ( __m128i a, __m128i b);
VPADDB __m256i _mm256_add_epi8 (__m256ia,__m256i b );
VPADDW __m256i _mm256_add_epi16 ( __m256i a, __m256i b);
VPADDD __m256i _mm256_add_epi32 ( __m256i a, __m256i b);
VPADDQ __m256i _mm256_add_epi64 ( __m256i a, __m256i b);
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ __m64 _mm_add_si64(__m64 m1, __m64 m2)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPADDD/Q, see Table 2-51, “Type E4 Class Exception Conditions.”

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EC /r[1] PADDSB mm, mm/m64	A	V/V	MMX	Add packed signed byte integers from mm/m64 and mm and saturate the results.
66 0F EC /r PADDSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed byte integers from xmm2/m128 and xmm1 saturate the results.
NP 0F ED /r[1] PADDSW mm, mm/m64	A	V/V	MMX	Add packed signed word integers from mm/m64 and mm and saturate the results.
66 0F ED /r PADDSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed signed word integers from xmm2/m128 and xmm1 and saturate the results.
VEX.128.66.0F.WIG EC /r VPADDSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed byte integers from xmm3/m128 and xmm2 saturate the results.
VEX.128.66.0F.WIG ED /r VPADDSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed signed word integers from xmm3/m128 and xmm2 and saturate the results.
VEX.256.66.0F.WIG EC /r VPADDSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
VEX.256.66.0F.WIG ED /r VPADDSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
EVEX.128.66.0F.WIG EC /r VPADDSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EC /r VPADDSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EC /r VPADDSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed signed byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG ED /r VPADDSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG ED /r VPADDSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG ED /r VPADDSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed signed word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDSB (With 64-bit Operands)

```
DEST[7:0] := SaturateToSignedByte(DEST[7:0] + SRC (7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] := SaturateToSignedByte(DEST[63:56] + SRC[63:56] );
```

PADDSB (With 128-bit Operands)

```
DEST[7:0] := SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

VPADDSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] := 0
```

VPADDSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] := SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDSB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PADDSW (with 64-bit operands)

```
    DEST[15:0] := SaturateToSignedWord(DEST[15:0] + SRC[15:0] );
    (* Repeat add operation for 2nd and 7th words *)
    DEST[63:48] := SaturateToSignedWord(DEST[63:48] + SRC[63:48] );
```

PADDSW (with 128-bit operands)

```
    DEST[15:0] := SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDSW (VEX.128 Encoded Version)

```
    DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[MAXVL-1:128] := 0
```

VPADDSW (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] := SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDSW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB __m64 _mm_adds_pi8(__m64 m1, __m64 m2)
(V)PADDSB __m128i _mm_adds_epi8 (__m128i a, __m128i b)
VPADDSB __m256i _mm256_adds_epi8 (__m256i a, __m256i b)
PADDSW __m64 _mm_adds_pi16(__m64 m1, __m64 m2)
(V)PADDSW __m128i _mm_adds_epi16 (__m128i a, __m128i b)
VPADDSW __m256i _mm256_adds_epi16 (__m256i a, __m256i b)
VPADDSB __m512i _mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW __m512i _mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB __m512i _mm512_mask_adds_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_mask_adds_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB __m512i _mm512_maskz_adds_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_maskz_adds_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDSB __m256i _mm256_mask_adds_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_mask_adds_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSB __m256i _mm256_maskz_adds_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_maskz_adds_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDSB __m128i _mm_mask_adds_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDSW __m128i _mm_mask_adds_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDSB __m128i _mm_maskz_adds_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDSW __m128i _mm_maskz_adds_epi16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PADDUSB/PADDUSW—Add Packed Unsigned Integers With Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DC /r[1] PADDUSB mm, mm/m64	A	V/V	MMX	Add packed unsigned byte integers from mm/m64 and mm and saturate the results.
66 0F DC /r PADDUSB xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results.
NP 0F DD /r[1] PADDUSW mm, mm/m64	A	V/V	MMX	Add packed unsigned word integers from mm/m64 and mm and saturate the results.
66 0F DD /r PADDUSW xmm1, xmm2/m128	A	V/V	SSE2	Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results.
VEX.128.660F.WIG DC /r VPADDUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned byte integers from xmm3/m128 to xmm2 and saturate the results.
VEX.128.66.0F.WIG DD /r VPADDUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results.
VEX.256.66.0F.WIG DC /r VPADDUSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
VEX.256.66.0F.WIG DD /r VPADDUSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.
EVEX.128.66.0F.WIG DC /r VPADDUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DC /r VPADDUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DC /r VPADDUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed unsigned byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG DD /r VPADDUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DD /r VPADDUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DD /r VPADDUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Add packed unsigned word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDUSB (With 64-bit Operands)

```
    DEST[7:0] := SaturateToUnsignedByte(DEST[7:0] + SRC (7:0] );
    (* Repeat add operation for 2nd through 7th bytes *)
    DEST[63:56] := SaturateToUnsignedByte(DEST[63:56] + SRC[63:56]
```

PADDUSB (With 128-bit Operands)

```
    DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
    (* Repeat add operation for 2nd through 14th bytes *)
    DEST[127:120] := SaturateToUnSignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDUSB (VEX.128 Encoded Version)

```
    DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
    (* Repeat subtract operation for 2nd through 14th bytes *)
    DEST[127:120] := SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
    DEST[MAXVL-1:128] := 0
```

VPADDUSB (VEX.256 Encoded Version)

```
    DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
    (* Repeat add operation for 2nd through 31st bytes *)
    DEST[255:248] := SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDUSW (With 64-bit Operands)

```
    DEST[15:0] := SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] := SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );
```

PADDUSW (With 128-bit Operands)

```
    DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToUnSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDUSW (VEX.128 Encoded Version)

```
    DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] := SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[MAXVL-1:128] := 0
```

VPADDUSW (VEX.256 Encoded Version)

```
    DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] := SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDUSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPADDUSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
PADDUSB __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
(V)PADDUSB __m128i _mm_adds_epu8 ( __m128i a, __m128i b)
(V)PADDUSW __m128i _mm_adds_epu16 ( __m128i a, __m128i b)
VPADDUSB __m256i _mm256_adds_epu8 ( __m256i a, __m256i b)
VPADDUSW __m256i _mm256_adds_epu16 ( __m256i a, __m256i b)
VPADDUSB __m512i _mm512_adds_epu8 ( __m512i a, __m512i b)
VPADDUSW __m512i _mm512_adds_epu16 ( __m512i a, __m512i b)
VPADDUSB __m512i _mm512_mask_adds_epu8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i _mm512_mask_adds_epu16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB __m512i _mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i _mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB __m256i _mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i _mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB __m256i _mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i _mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB __m128i _mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i _mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB __m128i _mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i _mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PALIGNR—Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A 0F /r ib[1] PALIGNR mm1, mm2/m64, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into mm1.
66 0F 3A 0F /r ib PALIGNR xmm1, xmm2/m128, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into xmm1.
VEX.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Concatenate xmm2 and xmm3/m128, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
VEX.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX2	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
EVEX.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.512.66.0F3A.WIG 0F /r ib VPALIGNR zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	C	V/V	AVX512BW OR AVX10.1	Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX, XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e., 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count

used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

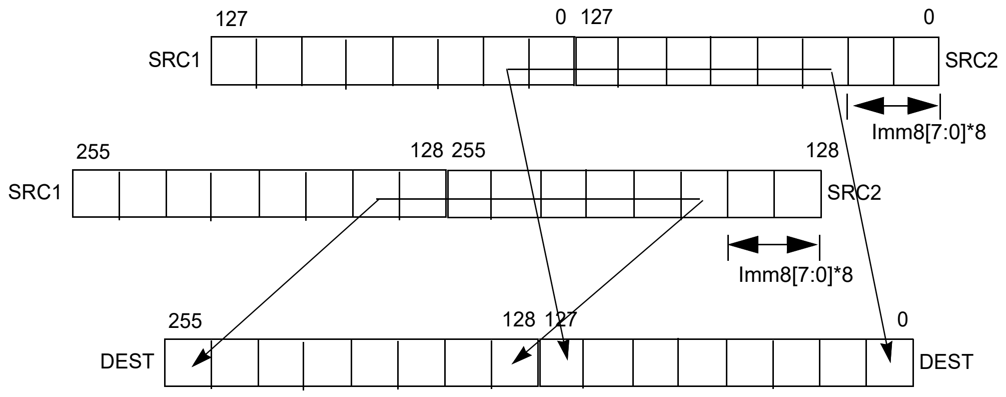


Figure 4-7. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR (With 64-bit Operands)

```
    temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
    DEST[63:0] = temp1[63:0]
```

PALIGNR (With 128-bit Operands)

```
temp1[255:0] := ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] := temp1[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

VPALIGNR (VEX.128 Encoded Version)

```
temp1[255:0] := ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] := temp1[127:0]
DEST[MAXVL-1:128] := 0
```

VPALIGNR (VEX.256 Encoded Version)

```
temp1[255:0] := ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] := temp1[127:0]
temp1[255:0] := ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAXVL-1:128] := temp1[127:0]
```

VPALIGNR (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR l := 0 TO VL-1 with increments of 128
    temp1[255:0] := ((SRC1[l+127:l] << 128) OR SRC2[l+127:l])>>(imm8[7:0]*8);
    TMP_DEST[l+127:l] := temp1[127:0]
ENDFOR;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
(V)PALIGNR __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
VPALIGNR __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)
VPALIGNR __m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)
VPALIGNR __m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DB /r[1] PAND mm, mm/m64	A	V/V	MMX	Bitwise AND mm/m64 and mm.
66 0F DB /r PAND xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND of xmm3/m128 and xmm.
VEX.256.66.0F.WIG DB /r VPAND ymm1, ymm2, ymm3/.m256	B	V/V	AVX2	Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.128.66.0F.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DB /r VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DB /r VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PAND (64-bit Operand)

```
DEST := DEST AND SRC
```

PAND (128-bit Legacy SSE Version)

```
DEST := DEST AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPAND (VEX.128 Encoded Version)

```
DEST := SRC1 AND SRC2
DEST[MAXVL-1:128] := 0
```

VPAND (VEX.256 Encoded Instruction)

```
DEST[255:0] := (SRC1[255:0] AND SRC2[255:0])
DEST[MAXVL-1:256] := 0
```

VPANDD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE AND SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPANDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPANDD __m512i _mm512_and_epi32( __m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_and_epi64( __m512i a, __m512i b);
VPANDQ __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_maskz_and_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_and_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_and_epi64( __mmask8 k, __m128i a, __m128i b);
PAND __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND __m128i _mm_and_si128 ( __m128i a, __m128i b)
VPAND  __m256i _mm256_and_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DF /r[1] PANDN mm, mm/m64	A	V/V	MMX	Bitwise AND NOT of mm/m64 and mm.
66 0F DF /r PANDN xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND NOT of xmm3/m128 and xmm2.
VEX.256.66.0F.WIG DF /r VPANDN ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.128.66.0F.W0 DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PANDN (64-bit Operand)

```
DEST := NOT(DEST) AND SRC
```

PANDN (128-bit Legacy SSE Version)

```
DEST := NOT(DEST) AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPANDN (VEX.128 Encoded Version)

```
DEST := NOT(SRC1) AND SRC2
DEST[MAXVL-1:128] := 0
```

VPANDN (VEX.256 Encoded Instruction)

```
DEST[255:0] := ((NOT SRC1[255:0]) AND SRC2[255:0])
DEST[MAXVL-1:256] := 0
```

VPANDND (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ((NOT SRC1[i+31:i]) AND SRC2[31:0])
                ELSE DEST[i+31:i] := ((NOT SRC1[i+31:i]) AND SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPANDNQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ((NOT SRC1[i+63:i]) AND SRC2[63:0])
                ELSE DEST[i+63:i] := ((NOT SRC1[i+63:i]) AND SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPANDND __m512i _mm512_andnot_epi32( __m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_andnot_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_andnot_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_andnot_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m512i _mm512_andnot_epi64( __m512i a, __m512i b);
VPANDNQ __m512i _mm512_mask_andnot_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m512i _mm512_maskz_andnot_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m256i _mm256_mask_andnot_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_andnot_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_andnot_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_andnot_epi64( __mmask8 k, __m128i a, __m128i b);
PANDN __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
(V)PANDN __m128i _mm_andnot_si128 ( __m128i a, __m128i b)
VPANDN __m256i _mm256_andnot_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E0 /r[1] PAVGB mm1, mm2/m64	A	V/V	SSE	Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.
66 0F E0, /r PAVGB xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.
NP 0F E3 /r[1] PAVGW mm1, mm2/m64	A	V/V	SSE	Average packed unsigned word integers from mm2/m64 and mm1 with rounding.
66 0F E3 /r PAVGW xmm1, xmm2/m128	A	V/V	SSE2	Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.
VEX.128.66.0F.WIG E0 /r VPAVGB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding.
VEX.128.66.0F.WIG E3 /r VPAVGW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding.
VEX.256.66.0F.WIG E0 /r VPAVGB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1.
VEX.256.66.0F.WIG E3 /r VPAVGW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1.
EVEX.128.66.0F.WIG E0 /r VPAVGB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned byte integers from xmm2, and xmm3/m128 with rounding and store to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E0 /r VPAVGB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E0 /r VPAVGB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Average packed unsigned byte integers from zmm2, and zmm3/m512 with rounding and store to zmm1 under writemask k1.
EVEX.128.66.0F.WIG E3 /r VPAVGW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned word integers from xmm2, xmm3/m128 with rounding to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E3 /r VPAVGW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E3 /r VPAVGW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Average packed unsigned word integers from zmm2, zmm3/m512 with rounding to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

Operation

PAVGB (With 64-bit Operands)

```
    DEST[7:0] := (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 6 *)
    DEST[63:56] := (SRC[63:56] + DEST[63:56] + 1) >> 1;
```

PAVGW (With 64-bit Operands)

```
    DEST[15:0] := (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 and 3 *)
    DEST[63:48] := (SRC[63:48] + DEST[63:48] + 1) >> 1;
```

PAVGB (With 128-bit Operands)

```
    DEST[7:0] := (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 14 *)
    DEST[127:120] := (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

PAVGW (With 128-bit Operands)

```
    DEST[15:0] := (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 through 6 *)
    DEST[127:112] := (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 Encoded Version)

```
    DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
    (* Repeat operation performed for bytes 2 through 15 *)
    DEST[127:120] := (SRC1[127:120] + SRC2[127:120] + 1) >> 1
    DEST[MAXVL-1:128] := 0
```

VPAVGW (VEX.128 Encoded Version)

```
    DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
    (* Repeat operation performed for 16-bit words 2 through 7 *)
    DEST[127:112] := (SRC1[127:112] + SRC2[127:112] + 1) >> 1
    DEST[MAXVL-1:128] := 0
```

VPAVGB (VEX.256 Encoded Instruction)

```
    DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 31)
    DEST[255:248] := (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

VPAVGW (VEX.256 Encoded Instruction)

```
    DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 through 15)
    DEST[255:14]) := (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
VPAVGB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPAVGW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1
                                            ; (* Temp sum before shifting is 17 bits *)
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);
PAVGB __m64 _mm_avg_pu8 (__m64 a, __m64 b)
PAVGW __m64 _mm_avg_pu16 (__m64 a, __m64 b)
(V)PAVGB __m128i _mm_avg_epu8 (__m128i a, __m128i b)
(V)PAVGW __m128i _mm_avg_epu16 (__m128i a, __m128i b)
VPAVGB __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
VPAVGW __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PCLMULQDQ—Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ xmm1, xmm2/m128, imm8	A	V/V	PCLMULQDQ	Performs carry-less multiplication of one quadword of xmm1 by one quadword of xmm2/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm1 and xmm2/m128 should be used.
VEX.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	B	V/V	PCLMULQDQ AVX	Performs carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.
VEX.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	B	V/V	VPCLMULQDQ AVX	For each 128-bit lane, performs two carry-less multiplications of one quadword of ymm2 by one quadword of ymm3/m256, stores the two 128-bit results in ymm1. The immediate is used to determine which quadword in each 128-bit lane of ymm2 and ymm3/m256 should be used.
EVEX.128.66.0F3A.WIG 44 /r /ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	C	V/V	VPCLMULQDQ (AVX512VL OR AVX10.1)	Performs carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.
EVEX.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	C	V/V	VPCLMULQDQ (AVX512VL OR AVX10.1)	For each 128-bit lane, performs two carry-less multiplications of one quadword of ymm2 by one quadword of ymm3/m256, stores the two 128-bit results in ymm1. The immediate is used to determine which quadword in each 128-bit lane of ymm2 and ymm3/m256 should be used.
EVEX.512.66.0F3A.WIG 44 /r /ib VPCLMULQDQ zmm1, zmm2, zmm3/m512, imm8	C	V/V	VPCLMULQDQ (AVX512F OR AVX10.1)	For each 128-bit lane, performs two carry-less multiplications of one quadword of zmm2 by one quadword of zmm3/m512, stores the four 128-bit results in zmm1. The immediate is used to determine which quadword in each 128-bit lane of zmm2 and zmm3/m512 should be used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Performs packed carry-less multiplication of quadword pairs. XMM versions perform a single multiply of a pair of quadwords. YMM versions perform two packed multiplies of pairs of quadwords. ZMM versions perform four packed multiplies of pairs of quadwords. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-13, other bits of the immediate byte are ignored.

The EVEX encoded form of this instruction does not support memory fault suppression.

Table 4-13. PCLMULQDQ Quadword Selection of Immediate Byte

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2[1][63:0], SRC1[63:0])
0	1	CL_MUL(SRC2[63:0], SRC1[127:64])
1	0	CL_MUL(SRC2[127:64], SRC1[63:0])
1	1	CL_MUL(SRC2[127:64], SRC1[127:64])

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. Bits (VL_MAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for imm8.

Table 4-14. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ xmm1, xmm2	**0000_0000B**
PCLMULHQLQDQ xmm1, xmm2	**0000_0001B**
PCLMULLQHQDQ xmm1, xmm2	**0001_0000B**
PCLMULHQHQDQ xmm1, xmm2	**0001_0001B**

Operation

```
define PCLMUL128(X,Y):                  // helper function
    FOR i := 0 to 63:
        TMP [ i ] := X[ 0 ] and Y[ i ]
        FOR j := 1 to i:
            TMP [ i ] := TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] := TMP[ i ]
    FOR i := 64 to 126:
        TMP [ i ] := 0
        FOR j := i - 63 to 63:
            TMP [ i ] := TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] := TMP[ i ]
    DEST[127] := 0;
    RETURN DEST                         // 128b vector
```

PCLMULQDQ (SSE Version)

```
IF imm8[0] = 0:
    TEMP1 := SRC1.qword[0]
ELSE:
    TEMP1 := SRC1.qword[1]
IF imm8[4] = 0:
    TEMP2 := SRC2.qword[0]
ELSE:
    TEMP2 := SRC2.qword[1]
DEST[127:0] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:128] (Unmodified)
```

VPCLMULQDQ (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i= 0 to KL-1:
    IF imm8[0] = 0:
        TEMP1 := SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 := SRC1.xmm[i].qword[1]
    IF imm8[4] = 0:
        TEMP2 := SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 := SRC2.xmm[i].qword[1]
    DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0
```

VPCLMULQDQ (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    IF imm8[0] = 0:
        TEMP1 := SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 := SRC1.xmm[i].qword[1]
    IF imm8[4] = 0:
        TEMP2 := SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 := SRC2.xmm[i].qword[1]
    DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)PCLMULQDQ __m128i  _mm_clmulepi64_si128 (__m128i, __m128i, const int)
VPCLMULQDQ __m256i _mm256_clmulepi64_epi128(__m256i, __m256i, const int);
VPCLMULQDQ __m512i _mm512_clmulepi64_epi128(__m512i, __m512i, const int);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally:

#UD	If VEX.L = 1.

EVEX-encoded: See Table 2-52, “Type E4NF Class Exception Conditions.”

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 74 /r[1] PCMPEQB mm, mm/m64	A	V/V	MMX	Compare packed bytes in mm/m64 and mm for equality.
66 0F 74 /r PCMPEQB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed bytes in xmm2/m128 and xmm1 for equality.
NP 0F 75 /r[1] PCMPEQW mm, mm/m64	A	V/V	MMX	Compare packed words in mm/m64 and mm for equality.
66 0F 75 /r PCMPEQW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed words in xmm2/m128 and xmm1 for equality.
NP 0F 76 /r[1] PCMPEQD mm, mm/m64	A	V/V	MMX	Compare packed doublewords in mm/m64 and mm for equality.
66 0F 76 /r PCMPEQD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed doublewords in xmm2/m128 and xmm1 for equality.
VEX.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed bytes in xmm3/m128 and xmm2 for equality.
VEX.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed words in xmm3/m128 and xmm2 for equality.
VEX.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed doublewords in xmm3/m128 and xmm2 for equality.
VEX.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.256.66.0F.WIG 75 /r VPCMPEQW ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed words in ymm3/m256 and ymm2 for equality.
VEX.256.66.0F.WIG 76 /r VPCMPEQD ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed doublewords in ymm3/m256 and ymm2 for equality.
EVEX.128.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.
EVEX.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW OR AVX10.1	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW OR AVX10.1	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQB (With 64-bit Operands)

```
    IF DEST[7:0] = SRC[7:0]
        THEN DEST[7:0) := FFH;
        ELSE DEST[7:0] := 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] = SRC[63:56]
        THEN DEST[63:56] := FFH;
        ELSE DEST[63:56] := 0; FI;
```

COMPARE_BYTES_EQUAL (SRC1, SRC2)

```
    IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] := FFH;
    ELSE DEST[7:0] := 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] := FFH;
    ELSE DEST[127:120] := 0; FI;
```

COMPARE_WORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] := FFFFH;
    ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] := FFFFH;
    ELSE DEST[127:112] := 0; FI;
```

COMPARE_DWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] := FFFFFFFFH;
    ELSE DEST[31:0] := 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 0; FI;
```

PCMPEQB (With 128-bit Operands)

```
DEST[127:0] := COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQB (VEX.128 Encoded Version)
```
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQB (VEX.256 Encoded Version)
```
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQB (EVEX Encoded Versions)
```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+7:i] == SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPEQW (With 64-bit Operands)
```
    IF DEST[15:0] = SRC[15:0]
        THEN DEST[15:0] := FFFFH;
        ELSE DEST[15:0] := 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] = SRC[63:48]
        THEN DEST[63:48] := FFFFH;
        ELSE DEST[63:48] := 0; FI;
```

PCMPEQW (With 128-bit Operands)
```
DEST[127:0] := COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQW (VEX.128 Encoded Version)
```
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQW (VEX.256 Encoded Version)
```
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+15:i] == SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0              ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPEQD (With 64-bit Operands)

```
    IF DEST[31:0] = SRC[31:0]
        THEN DEST[31:0] := FFFFFFFFH;
        ELSE DEST[31:0] := 0; FI;
    IF DEST[63:32] = SRC[63:32]
        THEN DEST[63:32] := FFFFFFFFH;
        ELSE DEST[63:32] := 0; FI;
```

PCMPEQD (With 128-bit Operands)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQD (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0
```

VPCMPEQD (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] = SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] = SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0              ; zeroing-masking only
    FI;
ENDFOR
```

```
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPCMPEQB __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask( __m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);
PCMPEQB __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW  __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB __m128i _mm_cmpeq_epi8 ( __m128i a, __m128i b)
(V)PCMPEQW __m128i _mm_cmpeq_epi16 ( __m128i a, __m128i b)
(V)PCMPEQD __m128i _mm_cmpeq_epi32 ( __m128i a, __m128i b)
VPCMPEQB __m256i _mm256_cmpeq_epi8 ( __m256i a, __m256i b)
VPCMPEQW __m256i _mm256_cmpeq_epi16 ( __m256i a, __m256i b)
VPCMPEQD __m256i _mm256_cmpeq_epi32 ( __m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPCMPEQD, see Table 2-51, “Type E4 Class Exception Conditions.”
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PCMPEQQ—Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed qwords in xmm2/m128 and xmm1 for equality.
VEX.128.66.0F38.WIG 29 /r VPCMPEQQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed quadwords in xmm3/m128 and xmm2 for equality.
VEX.256.66.0F38.WIG 29 /r VPCMPEQQ ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed quadwords in ymm3/m256 and ymm2 for equality.
EVEX.128.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Equal between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare Equal between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQQ (With 128-bit Operands)

```
IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
DEST[MAXVL-1:128] (Unmodified)
```

COMPARE_QWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
    IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
```

VPCMPEQQ (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPEQQ (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_QWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPEQQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] = SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] = SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPEQQ __mmask8 _mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm256_cmpeq_epi64_mask(__m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm256_mask_cmpeq_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm_cmpeq_epi64_mask(__m128i a, __m128i b);
VPCMPEQQ __mmask8 _mm_mask_cmpeq_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPCMPEQQ, see Table 2-51, “Type E4 Class Exception Conditions.”

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 64 /r[1] PCMPGTB mm, mm/m64	A	V/V	MMX	Compare packed signed byte integers in mm and mm/m64 for greater than.
66 0F 64 /r PCMPGTB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.
NP 0F 65 /r[1] PCMPGTW mm, mm/m64	A	V/V	MMX	Compare packed signed word integers in mm and mm/m64 for greater than.
66 0F 65 /r PCMPGTW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.
NP 0F 66 /r[1] PCMPGTD mm, mm/m64	A	V/V	MMX	Compare packed signed doubleword integers in mm and mm/m64 for greater than.
66 0F 66 /r PCMPGTD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.
VEX.128.66.0F.WIG 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than.
VEX.128.66.0F.WIG 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than.
VEX.128.66.0F.WIG 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than.
VEX.256.66.0F.WIG 64 /r VPCMPGTB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than.
VEX.256.66.0F.WIG 65 /r VPCMPGTW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than.
VEX.256.66.0F.WIG 66 /r VPCMPGTD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed doubleword integers in ymm2 and ymm3/m256 for greater than.
EVEX.128.66.0F.W0 66 /r VPCMPGTD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 66 /r VPCMPGTD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.128.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW OR AVX10.1	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	D	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (With 64-bit Operands)

```
    IF DEST[7:0] > SRC[7:0]
        THEN DEST[7:0) := FFH;
        ELSE DEST[7:0] := 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] > SRC[63:56]
        THEN DEST[63:56] := FFH;
        ELSE DEST[63:56] := 0; FI;
```

COMPARE_BYTES_GREATER (SRC1, SRC2)

```
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] := FFH;
    ELSE DEST[7:0] := 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] := FFH;
    ELSE DEST[127:120] := 0; FI;
```

COMPARE_WORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] := FFFFH;
    ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] := FFFFH;
    ELSE DEST[127:112] := 0; FI;
```

COMPARE_DWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] := FFFFFFFFH;
    ELSE DEST[31:0] := 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] := FFFFFFFFH;
    ELSE DEST[127:96] := 0; FI;
```

PCMPGTB (With 128-bit Operands)

```
DEST[127:0] := COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTB (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTB (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+7:i] > SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPGTW (With 64-bit Operands)

```
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] := FFFFH;
        ELSE DEST[15:0] := 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] > SRC[63:48]
        THEN DEST[63:48] := FFFFH;
        ELSE DEST[63:48] := 0; FI;
```

PCMPGTW (With 128-bit Operands)

```
DEST[127:0] := COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTW (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTW (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0              ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

PCMPGTD (With 64-bit Operands)

```
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] := FFFFFFFFH;
        ELSE DEST[31:0] := 0; FI;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] := FFFFFFFFH;
        ELSE DEST[63:32] := 0; FI;
```

PCMPGTD (With 128-bit Operands)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTD (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTD (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] > SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0              ; zeroing-masking only
    FI;
ENDFOR
```

DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

```
VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB __m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW __m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD __m64 _mm_cmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB __m128i _mm_cmpgt_epi8 ( __m128i a, __m128i b)
(V)PCMPGTW __m128i _mm_cmpgt_epi16 ( __m128i a, __m128i b)
(V)DCMPGTD __m128i _mm_cmpgt_epi32 ( __m128i a, __m128i b)
VPCMPGTB __m256i _mm256_cmpgt_epi8 ( __m256i a, __m256i b)
VPCMPGTW __m256i _mm256_cmpgt_epi16 ( __m256i a, __m256i b)
VPCMPGTD __m256i _mm256_cmpgt_epi32 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPCMPGTD, see Table 2-51, “Type E4 Class Exception Conditions.”

EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PCMPGTQ—Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ xmm1,xmm2/m128	A	V/V	SSE4_2	Compare packed signed qwords in xmm2/m128 and xmm1 for greater than.
VEX.128.66.0F38.WIG 37 /r VPCMPGTQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed qwords in xmm2 and xmm3/m128 for greater than.
VEX.256.66.0F38.WIG 37 /r VPCMPGTQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed qwords in ymm2 and ymm3/m256 for greater than.
EVEX.128.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

```
COMPARE_QWORDS_GREATER (SRC1, SRC2)
    IF SRC1[63:0] > SRC2[63:0]
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] := 0; FI;
    IF SRC1[127:64] > SRC2[127:64]
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] := 0; FI;
```

VPCMPGTQ (VEX.128 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0
```

VPCMPGTQ (VEX.256 Encoded Version)

```
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] := COMPARE_QWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] := 0
```

VPCMPGTQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] > SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] > SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask(__m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm256_cmpgt_epi64_mask(__m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm_cmpgt_epi64_mask(__m128i a, __m128i b);
VPCMPGTQ __mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPGTQ __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
VPCMPGTQ __m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPCMPGTQ, see Table 2-51, “Type E4 Class Exception Conditions.”

PEXTRB/PEXTRD/PEXTRQ—Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB reg/m8, xmm2, imm8	A	V/V	SSE4_1	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD r/m32, xmm2, imm8	A	V/V	SSE4_1	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r/m32.
66 REX.W 0F 3A 16 /r ib PEXTRQ r/m64, xmm2, imm8	A	V/N.E.	SSE4_1	Extract a qword integer value from xmm2 at the source qword offset specified by imm8 into r/m64.
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB reg/m8, xmm2, imm8	A	V[1]/V	AVX	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	A	V/V	AVX	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	A	V/I[2]	AVX	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB reg/m8, xmm2, imm8	B	V/V	AVX512BW OR AVX10.1	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	B	V/V	AVX512DQ OR AVX10.1	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	B	V/N.E.[2]	AVX512DQ OR AVX10.1	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L”L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
CASE of
    PEXTRB: SEL := COUNT[3:0];
            TEMP := (Src >> SEL*8) AND FFH;
            IF (DEST = Mem8)
                THEN
                Mem8 := TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] := TEMP[7:0];
                    r64[63:8] := ZERO_FILL; };
            ELSE
                    R32[7:0] := TEMP[7:0];
                    r32[31:8] := ZERO_FILL; };
            FI;
    PEXTRD:SEL := COUNT[1:0];
            TEMP := (Src >> SEL*32) AND FFFF_FFFFH;
            DEST := TEMP;
    PEXTRQ: SEL := COUNT[0];
            TEMP := (Src >> SEL*64);
            DEST := TEMP;
EASC:
```

VPEXTRTD/VPEXTRQ

```
IF (64-Bit Mode and 64-bit dest operand)
THEN
    Src_Offset := imm8[0]
    r64/m64 := (Src >> Src_Offset * 64)
ELSE
    Src_Offset := imm8[1:0]
    r32/m32 := ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
FI
```

VPEXTRB (dest=m8)

```
SRC_Offset := imm8[3:0]
Mem8 := (Src >> Src_Offset*8)
```

VPEXTRB (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset := imm8[3:0]
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] := ZERO_FILL;
ELSE
    SRC_Offset := imm8[3:0];
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] := ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PEXTRB int _mm_extract_epi8 (__m128i src, const int ndx);
PEXTRD int _mm_extract_epi32 (__m128i src, const int ndx);
PEXTRQ __int64 _mm_extract_epi64 (__m128i src, const int ndx);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C5 /r ib[1] PEXTRW reg, mm, imm8	A	V/V	SSE	Extract the word specified by imm8 from mm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /r ib PEXTRW reg, xmm, imm8	A	V/V	SSE2	Extract the word specified by imm8 from xmm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW reg/m16, xmm, imm8	B	V/V	SSE4_1	Extract the word specified by imm8 from xmm and copy it to lowest 16 bits of reg or m16. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW reg, xmm1, imm8	A	V[2]/V	AVX	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW reg/m16, xmm2, imm8	B	V/V	AVX	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW reg, xmm1, imm8	A	V/V	AVX512BW OR AVX10.1	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW reg/m16, xmm2, imm8	C	V/V	AVX512BW OR AVX10.1	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
IF (DEST = Mem16)
THEN
    SEL := COUNT[2:0];
    TEMP := (Src >> SEL*16) AND FFFFH;
    Mem16 := TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
        FOR (PEXTRW instruction with 64-bit source operand)
            { SEL := COUNT[1:0];
              TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
              r64[15:0] := TEMP[15:0];
              r64[63:16] := ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
            { SEL := COUNT[2:0];
              TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
              r64[15:0] := TEMP[15:0];
              r64[63:16] := ZERO_FILL; }
    ELSE
        FOR (PEXTRW instruction with 64-bit source operand)
            {   SEL := COUNT[1:0];
                TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
                r32[15:0] := TEMP[15:0];
                r32[31:16] := ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
            {   SEL := COUNT[2:0];
                TEMP := (SRC >> (SEL ∗ 16)) AND FFFFH;
                r32[15:0] := TEMP[15:0];
                r32[31:16] := ZERO_FILL; };
    FI;
FI;
```

VPEXTRW (dest=m16)

```
SRC_Offset := imm8[2:0]
Mem16 := (Src >> Src_Offset*16)
```

VPEXTRW (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] := ZERO_FILL;
ELSE
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] := ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW int _mm_extract_pi16 (__m64 a, int n)

PEXTRW int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PINSRB/PINSRD/PINSRQ—Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB xmm1, r32/m8, imm8	A	V/V	SSE4_1	Insert a byte integer value from r32/m8 into xmm1 at the destination element in xmm1 specified by imm8.
66 0F 3A 22 /r ib PINSRD xmm1, r/m32, imm8	A	V/V	SSE4_1	Insert a dword integer value from r/m32 into the xmm1 at the destination element specified by imm8.
66 REX.W 0F 3A 22 /r ib PINSRQ xmm1, r/m64, imm8	A	V/N. E.	SSE4_1	Insert a qword integer value from r/m64 into the xmm1 at the destination element specified by imm8.
VEX.128.66.0F3A.W0 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	B	V[1]/V	AVX	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
VEX.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r/m32, imm8	B	V/V	AVX	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
VEX.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r/m64, imm8	B	V/I[2]	AVX	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.
EVEX.128.66.0F3A.WIG 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	C	V/V	AVX512BW OR AVX10.1	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
EVEX.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r32/m32, imm8	C	V/V	AVX512DQ OR AVX10.1	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
EVEX.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8	C	V/N.E.[2]	AVX512DQ OR AVX10.1	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L’L must be 0, otherwise the instruction will #UD.

Operation

```
CASE OF
    PINSRB:  SEL := COUNT[3:0];
             MASK := (0FFH << (SEL * 8));
             TEMP := (((SRC[7:0] << (SEL *8)) AND MASK);
    PINSRD:  SEL := COUNT[1:0];
             MASK := (0FFFFFFFFH << (SEL * 32));
             TEMP := (((SRC << (SEL *32)) AND MASK)   ;
    PINSRQ:  SEL := COUNT[0]
             MASK := (0FFFFFFFFFFFFFFFFH << (SEL * 64));
             TEMP := (((SRC << (SEL *64)) AND MASK)   ;
ESAC;
        DEST := ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX/EVEX Encoded Version)

```
SEL := imm8[3:0]
DEST[127:0] := write_b_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

VPINSRD (VEX/EVEX Encoded Version)

```
SEL := imm8[1:0]
DEST[127:0] := write_d_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

VPINSRQ (VEX/EVEX Encoded Version)

```
SEL := imm8[0]
DEST[127:0] := write_q_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PINSRB __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C4 /r ib[1] PINSRW mm, r32/m16, imm8	A	V/V	SSE	Insert the low word from r32 or from m16 into mm at the word position specified by imm8.
66 0F C4 /r ib PINSRW xmm, r32/m16, imm8	A	V/V	SSE2	Move the low word of r32 or from m16 into xmm at the word position specified by imm8.
VEX.128.66.0F.W0 C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	B	V[2]/V	AVX	Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.
EVEX.128.66.0F.WIG C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	C	V/V	AVX512BW OR AVX10.1	Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Three operand MMX and SSE instructions:

Copies a word from the source operand and inserts it in the destination operand at the location specified with the count operand. (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

Four operand AVX and AVX-512 instructions:

Combines a word from the first source operand with the second source operand, and inserts it in the destination operand at the location specified with the count operand. The second source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The first source and destination operands are XMM registers. The count operand is an 8-bit immediate. When specifying a word location, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L/EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

PINSRW dest, src, imm8 (MMX)

```
    SEL := imm8[1:0]
    DEST.word[SEL] := src.word[0]
```

PINSRW dest, src, imm8 (SSE)

```
    SEL := imm8[2:0]
    DEST.word[SEL] := src.word[0]
```

VPINSRW dest, src1, src2, imm8 (AVX/AVX512)

```
    SEL := imm8[2:0]
    DEST := src1
    DEST.word[SEL] := src2.word[0]
    DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW __m64 _mm_insert_pi16 (__m64 a, int d, int n)
PINSRW __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-59, “Type E9NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 04 /r[1] PMADDUBSW mm1, mm2/m64	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to mm1.
66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1.
EVEX.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 04 /r VPMADDUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

PMADDUBSW (With 64-bit Operands)

```
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
    DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
    DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
    DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (With 128-bit Operands)

```
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
    // Repeat operation for 2nd through 7th word
    SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-112]* DEST[119-112]);
```

VPMADDUBSW (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] := SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[MAXVL-1:128] := 0
```

VPMADDUBSW (VEX.256 Encoded Version)

```
DEST[15:0] := SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] := SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAXVL-1:256] := 0
```

VPMADDUBSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMADDUBSW __m512i _mm512_maddubs_epi16( __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_maddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_maddubs_epi16( __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_maddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_maddubs_epi16( __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_maddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16( __mmask8 k, __m128i a, __m128i b);
PMADDUBSW __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r[1] PMADDWD mm, mm/m64	A	V/V	MMX	Multiply the packed words in mm by the packed words in mm/m64, add adjacent doubleword results, and store in mm.
66 0F F5 /r PMADDWD xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/m128, add adjacent doubleword results, and store in xmm1.
VEX.128.66.0F.WIG F5 /r VPMADDWD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1.
VEX.256.66.0F.WIG F5 /r VPMADDWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1.
EVEX.128.66.0F.WIG F5 /r VPMADDWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.
EVEX.256.66.0F.WIG F5 /r VPMADDWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.
EVEX.512.66.0F.WIG F5 /r VPMADDWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

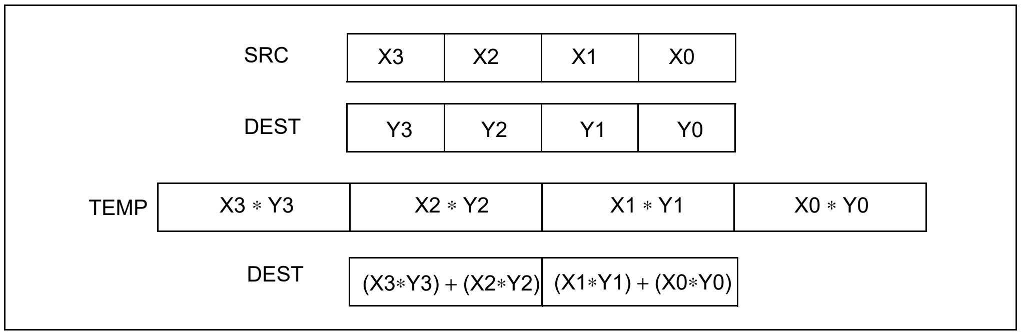


Figure 4-11. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (With 64-bit Operands)

```
    DEST[31:0] := (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
    DEST[63:32] := (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
```

PMADDWD (With 128-bit Operands)

```
    DEST[31:0] := (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
    DEST[63:32] := (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
    DEST[95:64] := (DEST[79:64] ∗ SRC[79:64]) + (DEST[95:80] ∗ SRC[95:80]);
    DEST[127:96] := (DEST[111:96] ∗ SRC[111:96]) + (DEST[127:112] ∗ SRC[127:112]);
```

VPMADDWD (VEX.128 Encoded Version)

```
DEST[31:0] := (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] := (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] := (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] := (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[MAXVL-1:128] := 0
```

VPMADDWD (VEX.256 Encoded Version)

```
DEST[31:0] := (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] := (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] := (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] := (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[159:128] := (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] := (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] := (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] := (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[MAXVL-1:256] := 0
```

VPMADDWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMADDWD __m512i _mm512_madd_epi16( __m512i a, __m512i b);
VPMADDWD __m512i _mm512_mask_madd_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m512i _mm512_maskz_madd_epi16( __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m256i _mm256_mask_madd_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m256i _mm256_maskz_madd_epi16( __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m128i _mm_mask_madd_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i _mm_maskz_madd_epi16( __mmask8 k, __m128i a, __m128i b);
PMADDWD __m64 _mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD __m128i _mm_madd_epi16 ( __m128i a, __m128i b)
VPMADDWD __m256i _mm256_madd_epi16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EE /r[1] PMAXSW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return maximum values.
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3C /r VPMAXSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EE /r VPMAXSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EE /r VPMAXSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EE /r VPMAXSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXSW (64-bit Operands)

```
    IF DEST[15:0] > SRC[15:0]) THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd and 3rd words in source and destination operands *)
    IF DEST[63:48] > SRC[63:48]) THEN
        DEST[63:48] := DEST[63:48];
    ELSE
        DEST[63:48] := SRC[63:48]; FI;
```

PMAXSB (128-bit Legacy SSE Version)

```
    IF DEST[7:0] > SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[7:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] > SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] > SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMAXSW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMAXSD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXSD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
                FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMAXSQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
                FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXSB __m512i _mm512_max_epi8(__m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8(__mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_max_epi16(__m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16(__mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8(__mmask32 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8(__mmask16 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m256i _mm256_mask_max_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32(__mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64(__mmask8 k, __m256i a, __m256i b);
VPMAXSD __m128i _mm_mask_max_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m512i _mm512_max_epi32(__m512i a, __m512i b);
VPMAXSD __m512i _mm512_mask_max_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXSD __m512i _mm512_maskz_max_epi32(__mmask16 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_max_epi64(__m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64(__mmask8 k, __m512i a, __m512i b);
(V)PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);
(V)PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)
(V)PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8 (__m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16 (__m256i a, __m256i b)
VPMAXSD __m256i _mm256_max_epi32 (__m256i a, __m256i b);
PMAXSW:__m64 _mm_max_pi16(__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPMAXSD/Q, see Table 2-51, “Type E4 Class Exception Conditions.”
EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DE /r[1] PMAXUB mm1, mm2/m64	A	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.
66 0F DE /r PMAXUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
VEX.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.256.66.0F DE /r VPMAXUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F38 3E/r VPMAXUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
EVEX.128.66.0F.WIG DE /r VPMAXUB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DE /r VPMAXUB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DE /r VPMAXUB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.WIG 3E /r VPMAXUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3E /r VPMAXUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3E /r VPMAXUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUB (64-bit Operands)

```
    IF DEST[7:0] > SRC[17:0]) THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[7:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 7th bytes in source and destination operands *)
    IF DEST[63:56] > SRC[63:56]) THEN
        DEST[63:56] := DEST[63:56];
    ELSE
        DEST[63:56] := SRC[63:56]; FI;
```

PMAXUB (128-bit Legacy SSE Version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[15:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMAXUW (128-bit Legacy SSE Version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMAXUB __m512i _mm512_max_epu8( __m512i a, __m512i b);
VPMAXUB __m512i _mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i _mm512_maskz_max_epu8( __mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_max_epu16( __m512i a, __m512i b);
VPMAXUW __m512i _mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_maskz_max_epu16( __mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i _mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i _mm256_maskz_max_epu8( __mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_maskz_max_epu16( __mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i _mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i _mm_maskz_max_epu8( __mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_maskz_max_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i _mm_max_epu8 ( __m128i a, __m128i b);
(V)PMAXUW __m128i _mm_max_epu16 ( __m128i a, __m128i b)
VPMAXUB __m256i _mm256_max_epu8 ( __m256i a, __m256i b);
VPMAXUW __m256i _mm256_max_epu16 ( __m256i a, __m256i b);
PMAXUB __m64 _mm_max_pu8(__m64 a, __m64 b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.W0 3F /r VPMAXUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3F /r VPMAXUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3F /r VPMAXUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3F /r VPMAXUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMAXUD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMAXUD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
                FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPMAXUQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
                FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUD __m512i _mm512_max_epu32(__m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_max_epu64(__m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
VPMAXUD __m256i _mm256_mask_max_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32(__mmask16 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b);
VPMAXUD __m128i _mm_mask_max_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_mask_max_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);
VPMAXUD __m256i _mm256_max_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EA /r[1] PMINSW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSW (64-bit Operands)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd and 3rd words in source and destination operands *)
    IF DEST[63:48] < SRC[63:48] THEN
        DEST[63:48] := DEST[63:48];
    ELSE
        DEST[63:48] := SRC[63:48]; FI;
```

PMINSB (128-bit Legacy SSE Version)

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[15:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSB (VEX.128 Encoded Version)

```
   IF SRC1[7:0] < SRC2[7:0] THEN
         DEST[7:0] := SRC1[7:0];
   ELSE
         DEST[7:0] := SRC2[7:0]; FI;
   (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
   IF SRC1[127:120] < SRC2[127:120] THEN
         DEST[127:120] := SRC1[127:120];
   ELSE
         DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSB (VEX.256 Encoded Version)

```
   IF SRC1[7:0] < SRC2[7:0] THEN
         DEST[7:0] := SRC1[7:0];
   ELSE
         DEST[15:0] := SRC2[7:0]; FI;
   (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
   IF SRC1[255:248] < SRC2[255:248] THEN
         DEST[255:248] := SRC1[255:248];
   ELSE
         DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
   i := j * 8
   IF k1[j] OR *no writemask* THEN
         IF SRC1[i+7:i] < SRC2[i+7:i]
               THEN DEST[i+7:i] := SRC1[i+7:i];
               ELSE DEST[i+7:i] := SRC2[i+7:i];
         FI;
         ELSE
               IF *merging-masking*                    ; merging-masking
                     THEN *DEST[i+7:i] remains unchanged*
                     ELSE                              ; zeroing-masking
                           DEST[i+7:i] := 0
               FI
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMINSW (128-bit Legacy SSE Version)

```
   IF DEST[15:0] < SRC[15:0] THEN
         DEST[15:0] := DEST[15:0];
   ELSE
         DEST[15:0] := SRC[15:0]; FI;
   (* Repeat operation for 2nd through 7th words in source and destination operands *)
   IF DEST[127:112] < SRC[127:112] THEN
         DEST[127:112] := DEST[127:112];
   ELSE
         DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSB __m512i _mm512_min_epi8(__m512i a, __m512i b);
VPMINSB __m512i _mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i _mm512_maskz_min_epi8(__mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16(__m512i a, __m512i b);
VPMINSW __m512i _mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi16(__mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i _mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i _mm256_maskz_min_epi8(__mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_maskz_min_epi16(__mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i _mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i _mm_maskz_min_epi8(__mmask16 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_maskz_min_epi16(__mmask8 k, __m128i a, __m128i b);
(V)PMINSB __m128i _mm_min_epi8 (__m128i a, __m128i b);
(V)PMINSW __m128i _mm_min_epi16 (__m128i a, __m128i b)
VPMINSB __m256i _mm256_min_epi8 (__m256i a, __m256i b);
VPMINSW __m256i _mm256_min_epi16 (__m256i a, __m256i b)
PMINSW__m64 _mm_min_pi16 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#MF	(64-bit operations only) If there is a pending x87 FPU exception.

PMINSD/PMINSQ—Minimum of Packed Signed Integers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSD (128-bit Legacy SSE Version)

```
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] := DEST[31:0];
    ELSE
        DEST[31:0] := SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] < SRC[127:96] THEN
        DEST[127:96] := DEST[127:96];
    ELSE
        DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSD (VEX.128 Encoded Version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] < SRC2[127:96] THEN
        DEST[127:96] := SRC1[127:96];
    ELSE
        DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINSD (VEX.256 Encoded Version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] := SRC1[31:0];
    ELSE
        DEST[31:0] := SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
        DEST[255:224] := SRC1[255:224];
    ELSE
        DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                             ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPMINSQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                             ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSD __m512i _mm512_min_epi32(__m512i a, __m512i b);
VPMINSD __m512i _mm512_mask_min_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSD __m512i _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_min_epi64(__m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSD __m256i _mm256_mask_min_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSD __m256i _mm256_maskz_min_epi32(__mmask16 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_mask_min_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_maskz_min_epi64(__mmask8 k, __m256i a, __m256i b);
VPMINSD __m128i _mm_mask_min_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSD __m128i _mm_maskz_min_epi32(__mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_mask_min_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);
VPMINSD __m256i _mm256_min_epi32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DA /r[1] PMINUB mm1, mm2/m64	A	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38 3A/r VPMINUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F DA /r VPMINUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F38 3A/r VPMINUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F DA /r VPMINUB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F DA /r VPMINUB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F DA /r VPMINUB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38 3A/r VPMINUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38 3A/r VPMINUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38 3A/r VPMINUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUB (64-bit Operands)

```
    IF DEST[7:0] < SRC[17:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[7:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 7th bytes in source and destination operands *)
    IF DEST[63:56] < SRC[63:56] THEN
        DEST[63:56] := DEST[63:56];
    ELSE
        DEST[63:56] := SRC[63:56]; FI;
```

PMINUB (128-bit Operands)

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] := DEST[7:0];
    ELSE
        DEST[15:0] := SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] := DEST[127:120];
    ELSE
        DEST[127:120] := SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUB (VEX.128 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[7:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] := SRC1[127:120];
    ELSE
        DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUB (VEX.256 Encoded Version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] := SRC1[7:0];
    ELSE
        DEST[15:0] := SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] := SRC1[255:248];
    ELSE
        DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] := SRC1[i+7:i];
            ELSE DEST[i+7:i] := SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

PMINUW (128-bit Operands)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] := DEST[15:0];
    ELSE
        DEST[15:0] := SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] := DEST[127:112];
    ELSE
        DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUW (VEX.128 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] := SRC1[127:112];
    ELSE
        DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUW (VEX.256 Encoded Version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] := SRC1[15:0];
    ELSE
        DEST[15:0] := SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] := SRC1[255:240];
    ELSE
        DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
            ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMINUB __m512i _mm512_min_epu8( __m512i a, __m512i b);
VPMINUB __m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i _mm512_maskz_min_epu8( __mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_min_epu16( __m512i a, __m512i b);
VPMINUW __m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_maskz_min_epu16( __mmask32 k, __m512i a, __m512i b);
VPMINUB __m256i _mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i _mm256_maskz_min_epu8( __mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_mask_min_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_maskz_min_epu16( __mmask16 k, __m256i a, __m256i b);
VPMINUB __m128i _mm_mask_min_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINUB __m128i _mm_maskz_min_epu8( __mmask16 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_mask_min_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_maskz_min_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i _mm_min_epu8 ( __m128i a, __m128i b)
(V)PMINUW __m128i _mm_min_epu16 ( __m128i a, __m128i b);
VPMINUB __m256i _mm256_min_epu8 ( __m256i a, __m256i b)
VPMINUW __m256i _mm256_min_epu16 ( __m256i a, __m256i b);
PMINUB __m64 _m_min_pu8 (__m64 a, __m64 b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 3B /r VPMINUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3B /r VPMINUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3B /r VPMINUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3B /r VPMINUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUD (128-bit Legacy SSE Version)

```
PMINUD instruction for 128-bit operands:
   IF DEST[31:0] < SRC[31:0] THEN
       DEST[31:0] := DEST[31:0];
   ELSE
       DEST[31:0] := SRC[31:0]; FI;
   (* Repeat operation for 2nd through 7th words in source and destination operands *)
   IF DEST[127:96] < SRC[127:96] THEN
       DEST[127:96] := DEST[127:96];
   ELSE
       DEST[127:96] := SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUD (VEX.128 Encoded Version)

```
VPMINUD instruction for 128-bit operands:
   IF SRC1[31:0] < SRC2[31:0] THEN
       DEST[31:0] := SRC1[31:0];
   ELSE
       DEST[31:0] := SRC2[31:0]; FI;
   (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
   IF SRC1[127:96] < SRC2[127:96] THEN
       DEST[127:96] := SRC1[127:96];
   ELSE
       DEST[127:96] := SRC2[127:96]; FI;
DEST[MAXVL-1:128] := 0
```

VPMINUD (VEX.256 Encoded Version)

```
VPMINUD instruction for 128-bit operands:
   IF SRC1[31:0] < SRC2[31:0] THEN
       DEST[31:0] := SRC1[31:0];
   ELSE
       DEST[31:0] := SRC2[31:0]; FI;
   (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
   IF SRC1[255:224] < SRC2[255:224] THEN
       DEST[255:224] := SRC1[255:224];
   ELSE
       DEST[255:224] := SRC2[255:224]; FI;
DEST[MAXVL-1:256] := 0
```

VPMINUD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                    ELSE DEST[i+31:i] := SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPMINUQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                    ELSE DEST[i+63:i] := SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMINUD __m512i _mm512_min_epu32( __m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_maskz_min_epu32( __mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_min_epu64( __m512i a, __m512i b);
VPMINUQ __m512i _mm512_mask_min_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64( __mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_mask_min_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUD __m256i _mm256_maskz_min_epu32( __mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64( __mmask8 k, __m256i a, __m256i b);
VPMINUD __m128i _mm_mask_min_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUD __m128i _mm_maskz_min_epu32( __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64( __mmask8 k, __m128i a, __m128i b);
(V)PMINUD __m128i _mm_min_epu32 ( __m128i a, __m128i b);
VPMINUD __m256i _mm256_min_epu32 ( __m256i a, __m256i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMOVSX—Packed Move With Sign Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	A	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.
EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW OR AVX10.1	Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F OR AVX10.1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 25 /r VPMOVSXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 25 /r VPMOVSXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

Packed_Sign_Extend_BYTE_to_WORD(DEST, SRC)

```
DEST[15:0] := SignExtend(SRC[7:0]);
DEST[31:16] := SignExtend(SRC[15:8]);
DEST[47:32] := SignExtend(SRC[23:16]);
DEST[63:48] := SignExtend(SRC[31:24]);
DEST[79:64] := SignExtend(SRC[39:32]);
DEST[95:80] := SignExtend(SRC[47:40]);
DEST[111:96] := SignExtend(SRC[55:48]);
DEST[127:112] := SignExtend(SRC[63:56]);
```

Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)

```
DEST[31:0] := SignExtend(SRC[7:0]);
DEST[63:32] := SignExtend(SRC[15:8]);
DEST[95:64] := SignExtend(SRC[23:16]);
DEST[127:96] := SignExtend(SRC[31:24]);
```

Packed_Sign_Extend_BYTE_to_QWORD(DEST, SRC)

```
DEST[63:0] := SignExtend(SRC[7:0]);
DEST[127:64] := SignExtend(SRC[15:8]);
```

Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)

```
DEST[31:0] := SignExtend(SRC[15:0]);
DEST[63:32] := SignExtend(SRC[31:16]);
DEST[95:64] := SignExtend(SRC[47:32]);
DEST[127:96] := SignExtend(SRC[63:48]);
```

Packed_Sign_Extend_WORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := SignExtend(SRC[15:0]);
DEST[127:64] := SignExtend(SRC[31:16]);
```

Packed_Sign_Extend_DWORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := SignExtend(SRC[31:0]);
DEST[127:64] := SignExtend(SRC[63:32]);
```

VPMOVSXBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXWQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVSXBW (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBD (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] := 0
```

VPMOVSXWD (VEX.256 Encoded Version)

```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXWQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVSXDQ (VEX.256 Encoded Version)

```
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVSXBW (VEX.128 Encoded Version)
```
Packed_Sign_Extend_BYTE_to_WORDDEST[127:0], SRC[127:0]()
DEST[MAXVL-1:128] := 0
```

VPMOVSXBD (VEX.128 Encoded Version)
```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXBQ (VEX.128 Encoded Version)
```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXWD (VEX.128 Encoded Version)
```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXWQ (VEX.128 Encoded Version)
```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

VPMOVSXDQ (VEX.128 Encoded Version)
```
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0
```

PMOVSXBW
```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXBD
```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXBQ
```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXWD
```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXWQ
```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

PMOVSXDQ
```
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMOVSXBW __m512i _mm512_cvtepi8_epi16(__m512i a);
VPMOVSXBW __m512i _mm512_mask_cvtepi8_epi16(__m512i a, __mmask32 k, __m512i b);
VPMOVSXBW __m512i _mm512_maskz_cvtepi8_epi16( __mmask32 k, __m512i b);
VPMOVSXBD __m512i _mm512_cvtepi8_epi32(__m512i a);
VPMOVSXBD __m512i _mm512_mask_cvtepi8_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXBD __m512i _mm512_maskz_cvtepi8_epi32( __mmask16 k, __m512i b);
VPMOVSXBQ __m512i _mm512_cvtepi8_epi64(__m512i a);
VPMOVSXBQ __m512i _mm512_mask_cvtepi8_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXBQ __m512i _mm512_maskz_cvtepi8_epi64( __mmask8 k, __m512i a);
VPMOVSXDQ __m512i _mm512_cvtepi32_epi64(__m512i a);
VPMOVSXDQ __m512i _mm512_mask_cvtepi32_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXDQ __m512i _mm512_maskz_cvtepi32_epi64( __mmask8 k, __m512i a);
VPMOVSXWD __m512i _mm512_cvtepi16_epi32(__m512i a);
VPMOVSXWD __m512i _mm512_mask_cvtepi16_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXWD __m512i _mm512_maskz_cvtepi16_epi32(__mmask16 k, __m512i a);
VPMOVSXWQ __m512i _mm512_cvtepi16_epi64(__m512i a);
VPMOVSXWQ __m512i _mm512_mask_cvtepi16_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXWQ __m512i _mm512_maskz_cvtepi16_epi64( __mmask8 k, __m512i a);
VPMOVSXBW __m256i _mm256_cvtepi8_epi16(__m256i a);
VPMOVSXBW __m256i  _mm256_mask_cvtepi8_epi16(__m256i a, __mmask16 k, __m256i b);
VPMOVSXBW __m256i _mm256_maskz_cvtepi8_epi16( __mmask16 k, __m256i b);
VPMOVSXBD __m256i _mm256_cvtepi8_epi32(__m256i a);
VPMOVSXBD __m256i _mm256_mask_cvtepi8_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBD __m256i _mm256_maskz_cvtepi8_epi32( __mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_cvtepi8_epi64(__m256i a);
VPMOVSXBQ __m256i _mm256_mask_cvtepi8_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_maskz_cvtepi8_epi64( __mmask8 k, __m256i a);
VPMOVSXDQ __m256i _mm256_cvtepi32_epi64(__m256i a);
VPMOVSXDQ __m256i _mm256_mask_cvtepi32_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXDQ __m256i _mm256_maskz_cvtepi32_epi64( __mmask8 k, __m256i a);
VPMOVSXWD __m256i _mm256_cvtepi16_epi32(__m256i a);
VPMOVSXWD __m256i _mm256_mask_cvtepi16_epi32(__m256i a, __mmask16 k, __m256i b);
VPMOVSXWD __m256i _mm256_maskz_cvtepi16_epi32(__mmask16 k, __m256i a);
VPMOVSXWQ __m256i _mm256_cvtepi16_epi64(__m256i a);
VPMOVSXWQ __m256i _mm256_mask_cvtepi16_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXWQ __m256i _mm256_maskz_cvtepi16_epi64( __mmask8 k, __m256i a);
VPMOVSXBW __m128i  _mm_mask_cvtepi8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBW __m128i _mm_maskz_cvtepi8_epi16( __mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_mask_cvtepi8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_maskz_cvtepi8_epi32( __mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_mask_cvtepi8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_maskz_cvtepi8_epi64( __mmask8 k, __m128i a);
VPMOVSXDQ __m128i _mm_mask_cvtepi32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXDQ __m128i _mm_maskz_cvtepi32_epi64( __mmask8 k, __m128i a);
VPMOVSXWD __m128i _mm_mask_cvtepi16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVSXWD __m128i _mm_maskz_cvtepi16_epi32(__mmask16 k, __m128i a);
VPMOVSXWQ __m128i _mm_mask_cvtepi16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXWQ __m128i _mm_maskz_cvtepi16_epi64( __mmask8 k, __m128i a);
PMOVSXBW __m128i _mm_ cvtepi8_epi16 ( __m128i a);
PMOVSXBD __m128i _mm_ cvtepi8_epi32 ( __m128i a);
PMOVSXBQ __m128i _mm_ cvtepi8_epi64 ( __m128i a);
PMOVSXWD __m128i _mm_ cvtepi16_epi32 ( __m128i a);
PMOVSXWQ __m128i _mm_ cvtepi16_epi64 ( __m128i a);
PMOVSXDQ __m128i _mm_ cvtepi32_epi64 ( __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-53, “Type E5 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PMOVZX—Packed Move With Zero Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 31 /r PMOVZXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F 38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	A	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38 30.WIG /r VPMOVZXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW OR AVX10.1	Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F OR AVX10.1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVZXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F OR AVX10.1	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 35 /r VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 35 /r VPMOVZXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F OR AVX10.1	Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Legacy, VEX, and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask.The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

```
Packed_Zero_Extend_BYTE_to_WORD(DEST, SRC)
DEST[15:0] := ZeroExtend(SRC[7:0]);
DEST[31:16] := ZeroExtend(SRC[15:8]);
DEST[47:32] := ZeroExtend(SRC[23:16]);
DEST[63:48] := ZeroExtend(SRC[31:24]);
DEST[79:64] := ZeroExtend(SRC[39:32]);
DEST[95:80] := ZeroExtend(SRC[47:40]);
DEST[111:96] := ZeroExtend(SRC[55:48]);
DEST[127:112] := ZeroExtend(SRC[63:56]);

Packed_Zero_Extend_BYTE_to_DWORD(DEST, SRC)
DEST[31:0] := ZeroExtend(SRC[7:0]);
DEST[63:32] := ZeroExtend(SRC[15:8]);
```

```
DEST[95:64] := ZeroExtend(SRC[23:16]);
DEST[127:96] := ZeroExtend(SRC[31:24]);
```

Packed_Zero_Extend_BYTE_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[7:0]);
DEST[127:64] := ZeroExtend(SRC[15:8]);
```

Packed_Zero_Extend_WORD_to_DWORD(DEST, SRC)

```
DEST[31:0] := ZeroExtend(SRC[15:0]);
DEST[63:32] := ZeroExtend(SRC[31:16]);
DEST[95:64] := ZeroExtend(SRC[47:32]);
DEST[127:96] := ZeroExtend(SRC[63:48]);
```

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[15:0]);
DEST[127:64] := ZeroExtend(SRC[31:16]);
```

Packed_Zero_Extend_DWORD_to_QWORD(DEST, SRC)

```
DEST[63:0] := ZeroExtend(SRC[31:0]);
DEST[127:64] := ZeroExtend(SRC[63:32]);
```

VPMOVZXBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXWD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXWQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVZXBW (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBD (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] := 0
```

VPMOVZXWD (VEX.256 Encoded Version)

```
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXWQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0
```

VPMOVZXDQ (VEX.256 Encoded Version)

```
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0
```

VPMOVZXBW (VEX.128 Encoded Version)
```
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXBD (VEX.128 Encoded Version)
```
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXBQ (VEX.128 Encoded Version)
```
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXWD (VEX.128 Encoded Version)
```
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXWQ (VEX.128 Encoded Version)
```
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] := 0
```

VPMOVZXDQ (VEX.128 Encoded Version)
```
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] := 0
```

PMOVZXBW
```
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXBD
```
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXBQ
```
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXWD
```
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXWQ
```
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)
```

PMOVZXDQ
```
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMOVZXBW __m512i _mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m512i _mm512_mask_cvtepu8_epi16(__m512i a, __mmask32 k, __m256i b);
VPMOVZXBW __m512i _mm512_maskz_cvtepu8_epi16( __mmask32 k, __m256i b);
VPMOVZXBD __m512i _mm512_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m512i _mm512_mask_cvtepu8_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXBD __m512i _mm512_maskz_cvtepu8_epi32( __mmask16 k, __m128i b);
VPMOVZXBQ __m512i _mm512_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m512i _mm512_mask_cvtepu8_epi64(__m512i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m512i _mm512_maskz_cvtepu8_epi64( __mmask8 k, __m128i a);
VPMOVZXDQ __m512i _mm512_cvtepu32_epi64(__m256i a);
VPMOVZXDQ __m512i _mm512_mask_cvtepu32_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXDQ __m512i _mm512_maskz_cvtepu32_epi64( __mmask8 k, __m256i a);
VPMOVZXWD __m512i _mm512_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m512i _mm512_mask_cvtepu16_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXWD __m512i _mm512_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m512i _mm512_cvtepu16_epi64(__m256i a);
VPMOVZXWQ __m512i _mm512_mask_cvtepu16_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXWQ __m512i _mm512_maskz_cvtepu16_epi64( __mmask8 k, __m256i a);
VPMOVZXBW __m256i _mm256_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m256i  _mm256_mask_cvtepu8_epi16(__m256i a, __mmask16 k, __m128i b);
VPMOVZXBW __m256i _mm256_maskz_cvtepu8_epi16( __mmask16 k, __m128i b);
VPMOVZXBD __m256i _mm256_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m256i _mm256_mask_cvtepu8_epi32(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBD __m256i _mm256_maskz_cvtepu8_epi32( __mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m256i _mm256_mask_cvtepu8_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_maskz_cvtepu8_epi64( __mmask8 k, __m128i a);
VPMOVZXDQ __m256i _mm256_cvtepu32_epi64(__m128i a);
VPMOVZXDQ __m256i _mm256_mask_cvtepu32_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXDQ __m256i _mm256_maskz_cvtepu32_epi64( __mmask8 k, __m128i a);
VPMOVZXWD __m256i _mm256_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m256i _mm256_mask_cvtepu16_epi32(__m256i a, __mmask16 k, __m128i b);
VPMOVZXWD __m256i _mm256_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m256i _mm256_cvtepu16_epi64(__m128i a);
VPMOVZXWQ __m256i _mm256_mask_cvtepu16_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m256i _mm256_maskz_cvtepu16_epi64( __mmask8 k, __m128i a);
VPMOVZXBW __m128i  _mm_mask_cvtepu8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBW __m128i _mm_maskz_cvtepu8_epi16( __mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_mask_cvtepu8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi32( __mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_mask_cvtepu8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_maskz_cvtepu8_epi64( __mmask8 k, __m128i a);
VPMOVZXDQ __m128i _mm_mask_cvtepu32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXDQ __m128i _mm_maskz_cvtepu32_epi64( __mmask8 k, __m128i a);
VPMOVZXWD __m128i _mm_mask_cvtepu16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVZXWD __m128i _mm_maskz_cvtepu16_epi32(__mmask8 k, __m128i a);
VPMOVZXWQ __m128i _mm_mask_cvtepu16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m128i _mm_maskz_cvtepu16_epi64( __mmask8 k, __m128i a);
PMOVZXBW __m128i _mm_ cvtepu8_epi16 ( __m128i a);
PMOVZXBD __m128i _mm_ cvtepu8_epi32 ( __m128i a);
PMOVZXBQ __m128i _mm_ cvtepu8_epi64 ( __m128i a);
PMOVZXWD __m128i _mm_ cvtepu16_epi32 ( __m128i a);
PMOVZXWQ __m128i _mm_ cvtepu16_epi64 ( __m128i a);
PMOVZXDQ __m128i _mm_ cvtepu32_epi64 ( __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-53, “Type E5 Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.128.66.0F38.W1 28 /r VPMULDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 28 /r VPMULDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

VPMULDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[31:0])
                ELSE DEST[i+63:i] := SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMULDQ (VEX.256 Encoded Version)

```
DEST[63:0] := SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] := SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[191:128] := SignExtend64( SRC1[159:128]) * SignExtend64( SRC2[159:128])
DEST[255:192] := SignExtend64( SRC1[223:192]) * SignExtend64( SRC2[223:192])
DEST[MAXVL-1:256] := 0
```

VPMULDQ (VEX.128 Encoded Version)

```
DEST[63:0] := SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] := SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[MAXVL-1:128] := 0
```

PMULDQ (128-bit Legacy SSE Version)

```
DEST[63:0] := SignExtend64( DEST[31:0]) * SignExtend64( SRC[31:0])
DEST[127:64] := SignExtend64( DEST[95:64]) * SignExtend64( SRC[95:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULDQ __m512i _mm512_mul_epi32(__m512i a, __m512i b);
VPMULDQ __m512i _mm512_mask_mul_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULDQ __m512i _mm512_maskz_mul_epi32(__mmask8 k, __m512i a, __m512i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__mmask8 k, __m128i a, __m128i b);
(V)PMULDQ __m128i _mm_mul_epi32(__m128i a, __m128i b);
VPMULDQ __m256i _mm256_mul_epi32(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMULHRSW—Packed Multiply High With Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 0B /r[1] PMULHRSW mm1, mm2/m64	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to mm1.
66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.
VEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1.
EVEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 0B /r VPMULHRSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHRSW (With 64-bit Operands)

```
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRc[63:48]) >> 14) + 1;
    DEST[15:0] = temp0[16:1];
    DEST[31:16] = temp1[16:1];
    DEST[47:32] = temp2[16:1];
    DEST[63:48] = temp3[16:1];
```

PMULHRSW (With 128-bit Operands)

```
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
    temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
    temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
    temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
    temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1;
    DEST[15:0] = temp0[16:1];
    DEST[31:16] = temp1[16:1];
    DEST[47:32] = temp2[16:1];
    DEST[63:48] = temp3[16:1];
    DEST[79:64] = temp4[16:1];
    DEST[95:80] = temp5[16:1];
    DEST[111:96] = temp6[16:1];
    DEST[127:112] = temp7[16:1];
```

VPMULHRSW (VEX.128 Encoded Version)

```
temp0[31:0] := INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] := INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] := INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] := INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] := INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] := INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] := INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] := INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] := temp0[16:1]
DEST[31:16] := temp1[16:1]
DEST[47:32] := temp2[16:1]
```

```
DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[MAXVL-1:128] := 0
```

VPMULHRSW (VEX.256 Encoded Version)

```
temp0[31:0] := INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] := INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] := INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] := INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] := INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] := INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] := INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] := INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
temp8[31:0] := INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] := INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] := INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] := INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] := INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] := INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] := INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] := INT32 ((SRC1[255:240] * SRC2[255:240) >>14) + 1

DEST[15:0] := temp0[16:1]
DEST[31:16] := temp1[16:1]
DEST[47:32] := temp2[16:1]
DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[143:128] := temp8[16:1]
DEST[159:144] := temp9[16:1]
DEST[175:160] := temp10[16:1]
DEST[191:176] := temp11[16:1]
DEST[207:192] := temp12[16:1]
DEST[223:208] := temp13[16:1]
DEST[239:224] := temp14[16:1]
DEST[255:240] := temp15[16:1]
DEST[MAXVL-1:256] := 0
```

VPMULHRSW (EVEX Encoded Version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
            DEST[i+15:i] := tmp[16:1]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i _mm_maskz_mulhrs_epi16( __mmask8 k, __m128i a, __m128i b);
PMULHRSW __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW __m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E4 /r[1] PMULHUW mm1, mm2/m64	A	V/V	SSE	Multiply the packed unsigned word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 0F E4 /r PMULHUW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.128.66.0F.WIG E4 /r VPMULHUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.256.66.0F.WIG E4 /r VPMULHUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG E4 /r VPMULHUW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E4 /r VPMULHUW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E4 /r VPMULHUW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

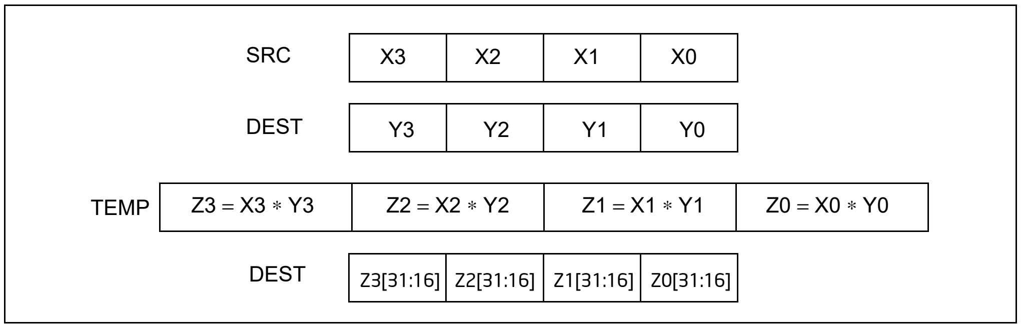


Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (With 64-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
    TEMP1[31:0] := DEST[31:16] * SRC[31:16];
    TEMP2[31:0] := DEST[47:32] * SRC[47:32];
    TEMP3[31:0] := DEST[63:48] * SRC[63:48];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
```

PMULHUW (With 128-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
    TEMP1[31:0] := DEST[31:16] * SRC[31:16];
    TEMP2[31:0] := DEST[47:32] * SRC[47:32];
    TEMP3[31:0] := DEST[63:48] * SRC[63:48];
    TEMP4[31:0] := DEST[79:64] * SRC[79:64];
    TEMP5[31:0] := DEST[95:80] * SRC[95:80];
    TEMP6[31:0] := DEST[111:96] * SRC[111:96];
    TEMP7[31:0] := DEST[127:112] * SRC[127:112];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
    DEST[79:64] := TEMP4[31:16];
    DEST[95:80] := TEMP5[31:16];
    DEST[111:96] := TEMP6[31:16];
    DEST[127:112] := TEMP7[31:16];
```

VPMULHUW (VEX.128 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0
```

PMULHUW (VEX.256 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] := SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] := SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] := SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] := SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] := SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] := SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] := SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] := SRC1[255:240] * SRC2[255:240]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[191:176] := TEMP11[31:16]
DEST[207:192] := TEMP12[31:16]
DEST[223:208] := TEMP13[31:16]
DEST[239:224] := TEMP14[31:16]
DEST[255:240] := TEMP15[31:16]
DEST[MAXVL-1:256] := 0
```

PMULHUW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16( __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16( __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i _mm_maskz_mulhi_epu16( __mmask8 k, __m128i a, __m128i b);
PMULHUW __m64 _mm_mulhi_pu16(__m64 a, __m64 b)
(V)PMULHUW __m128i _mm_mulhi_epu16 ( __m128i a, __m128i b)
VPMULHUW __m256i _mm256_mulhi_epu16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E5 /r[1] PMULHW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 0F E5 /r PMULHW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.128.66.0F.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.256.66.0F.WIG E5 /r VPMULHW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG E5 /r VPMULHW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E5 /r VPMULHW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E5 /r VPMULHW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

n 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHW (With 64-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
```

PMULHW (With 128-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] := DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] := DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] := DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] := DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] := TEMP0[31:16];
    DEST[31:16] := TEMP1[31:16];
    DEST[47:32] := TEMP2[31:16];
    DEST[63:48] := TEMP3[31:16];
    DEST[79:64] := TEMP4[31:16];
    DEST[95:80] := TEMP5[31:16];
    DEST[111:96] := TEMP6[31:16];
    DEST[127:112] := TEMP7[31:16];
```

VPMULHW (VEX.128 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
```

```
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0
```

PMULHW (VEX.256 Encoded Version)

```
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] := SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] := SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] := SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] := SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] := SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] := SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] := SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] := SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] := SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] := SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] := SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] := SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] := SRC1[255:240] * SRC2[255:240]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[47:32] := TEMP2[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[191:176] := TEMP11[31:16]
DEST[207:192] := TEMP12[31:16]
DEST[223:208] := TEMP13[31:16]
DEST[239:224] := TEMP14[31:16]
DEST[255:240] := TEMP15[31:16]
DEST[MAXVL-1:256] := 0
```

PMULHW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULHW __m512i _mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i _mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i _mm512_maskz_mulhi_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i _mm256_mask_mulhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i _mm256_maskz_mulhi_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i _mm_mask_mulhi_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i _mm_maskz_mulhi_epi16( __mmask8 k, __m128i a, __m128i b);
PMULHW __m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
(V)PMULHW __m128i _mm_mulhi_epi16 ( __m128i a, __m128i b)
VPMULHW __m256i _mm256_mulhi_epi16 ( __m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.
VEX.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.
EVEX.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

VPMULLQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN Temp[127:0] := SRC1[i+63:i] * SRC2[63:0]
                ELSE Temp[127:0] := SRC1[i+63:i] * SRC2[i+63:i]
            FI;
            DEST[i+63:i] := Temp[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMULLD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN Temp[63:0] := SRC1[i+31:i] * SRC2[31:0]
                ELSE Temp[63:0] := SRC1[i+31:i] * SRC2[i+31:i]
            FI;
            DEST[i+31:i] := Temp[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMULLD (VEX.256 Encoded Version)

```
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp1[63:0] := SRC1[63:32] * SRC2[63:32]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]
Temp3[63:0] := SRC1[127:96] * SRC2[127:96]
Temp4[63:0] := SRC1[159:128] * SRC2[159:128]
Temp5[63:0] := SRC1[191:160] * SRC2[191:160]
Temp6[63:0] := SRC1[223:192] * SRC2[223:192]
Temp7[63:0] := SRC1[255:224] * SRC2[255:224]

DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[159:128] := Temp4[31:0]
DEST[191:160] := Temp5[31:0]
DEST[223:192] := Temp6[31:0]
DEST[255:224] := Temp7[31:0]
DEST[MAXVL-1:256] := 0
```

VPMULLD (VEX.128 Encoded Version)

```
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp1[63:0] := SRC1[63:32] * SRC2[63:32]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]
Temp3[63:0] := SRC1[127:96] * SRC2[127:96]
DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] := 0
```

PMULLD (128-bit Legacy SSE Version)

```
Temp0[63:0] := DEST[31:0] * SRC[31:0]
Temp1[63:0] := DEST[63:32] * SRC[63:32]
Temp2[63:0] := DEST[95:64] * SRC[95:64]
Temp3[63:0] := DEST[127:96] * SRC[127:96]
DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask16 k, __m512i a, __m512i b);
VPMULLD __m256i _mm256_mask_mullo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLD __m128i _mm_mask_mullo_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLD __m128i _mm_maskz_mullo_epi32(__mmask8 k, __m128i a, __m128i b);
VPMULLD __m256i _mm256_mullo_epi32(__m256i a, __m256i b);
PMULLD __m128i _mm_mullo_epi32(__m128i a, __m128i b);
VPMULLQ __m512i _mm512_mullo_epi64(__m512i a, __m512i b);
VPMULLQ __m512i _mm512_mask_mullo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULLQ __m512i _mm512_maskz_mullo_epi64(__mmask8 k, __m512i a, __m512i b);
VPMULLQ __m256i _mm256_mullo_epi64(__m256i a, __m256i b);
VPMULLQ __m256i _mm256_mask_mullo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m256i _mm256_maskz_mullo_epi64(__mmask8 k, __m256i a, __m256i b);
VPMULLQ __m128i _mm_mullo_epi64(__m128i a, __m128i b);
VPMULLQ __m128i _mm_mask_mullo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLQ __m128i _mm_maskz_mullo_epi64(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D5 /r[1] PMULLW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the low 16 bits of the results in mm1.
66 0F D5 /r PMULLW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1.
VEX.128.66.0F.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.256.66.0F.WIG D5 /r VPMULLW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1.
EVEX.128.66.0F.WIG D5 /r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG D5 /r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG D5 /r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

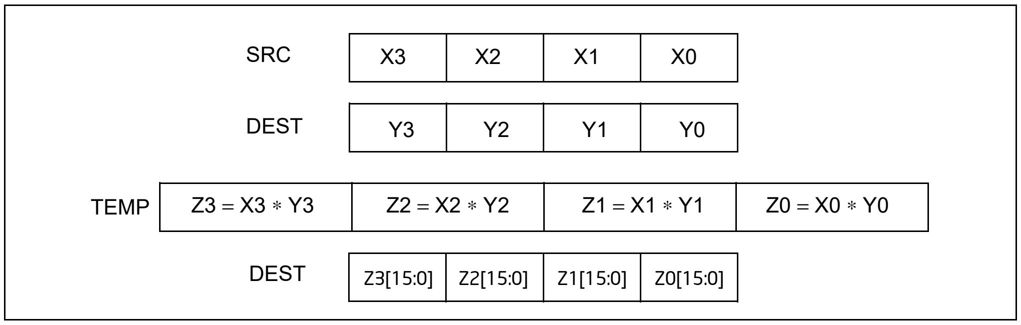


Figure 4-13. PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (With 64-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    DEST[15:0] := TEMP0[15:0];
    DEST[31:16] := TEMP1[15:0];
    DEST[47:32] := TEMP2[15:0];
    DEST[63:48] := TEMP3[15:0];
```

PMULLW (With 128-bit Operands)

```
    TEMP0[31:0] := DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] := DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] := DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] := DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] := DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] := DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] := DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] := DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] := TEMP0[15:0];
    DEST[31:16] := TEMP1[15:0];
    DEST[47:32] := TEMP2[15:0];
    DEST[63:48] := TEMP3[15:0];
    DEST[79:64] := TEMP4[15:0];
    DEST[95:80] := TEMP5[15:0];
    DEST[111:96] := TEMP6[15:0];
    DEST[127:112] := TEMP7[15:0];
DEST[MAXVL-1:256] := 0
```

VPMULLW (VEX.128 Encoded Version)

```
Temp0[31:0] := SRC1[15:0] * SRC2[15:0]
Temp1[31:0] := SRC1[31:16] * SRC2[31:16]
Temp2[31:0] := SRC1[47:32] * SRC2[47:32]
Temp3[31:0] := SRC1[63:48] * SRC2[63:48]
Temp4[31:0] := SRC1[79:64] * SRC2[79:64]
Temp5[31:0] := SRC1[95:80] * SRC2[95:80]
Temp6[31:0] := SRC1[111:96] * SRC2[111:96]
Temp7[31:0] := SRC1[127:112] * SRC2[127:112]
DEST[15:0] := Temp0[15:0]
DEST[31:16] := Temp1[15:0]
DEST[47:32] := Temp2[15:0]
DEST[63:48] := Temp3[15:0]
DEST[79:64] := Temp4[15:0]
DEST[95:80] := Temp5[15:0]
DEST[111:96] := Temp6[15:0]
DEST[127:112] := Temp7[15:0]
DEST[MAXVL-1:128] := 0
```

PMULLW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] := temp[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mask_mullo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm_mask_mullo_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm_maskz_mullo_epi16(__mmask8 k, __m128i a, __m128i b);
PMULLW __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW __m128i _mm_mullo_epi16 (__m128i a, __m128i b)
VPMULLW __m256i _mm256_mullo_epi16 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F4 /r[1] PMULUDQ mm1, mm2/m64	A	V/V	SSE2	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.128.66.0F.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.256.66.0F.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth, and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth, and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth, and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

PMULUDQ (With 64-Bit Operands)

```
    DEST[63:0] := DEST[31:0] ∗ SRC[31:0];
```

PMULUDQ (With 128-Bit Operands)

```
    DEST[63:0] := DEST[31:0] ∗ SRC[31:0];
    DEST[127:64] := DEST[95:64] ∗ SRC[95:64];
```

VPMULUDQ (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:64] := SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:128] := 0
```

VPMULUDQ (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[31:0] * SRC2[31:0]
DEST[127:64] := SRC1[95:64] * SRC2[95:64
DEST[191:128] := SRC1[159:128] * SRC2[159:128]
DEST[255:192] := SRC1[223:192] * SRC2[223:192]
DEST[MAXVL-1:256] := 0
```

VPMULUDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[31:0] )
                ELSE DEST[i+63:i] := ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31:i] )
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULUDQ __m512i _mm512_mul_epu32(__m512i a, __m512i b);
VPMULUDQ __m512i _mm512_mask_mul_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i _mm512_maskz_mul_epu32( __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i _mm256_mask_mul_epu32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i _mm256_maskz_mul_epu32( __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i _mm_mask_mul_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULUDQ __m128i _mm_maskz_mul_epu32( __mmask8 k, __m128i a, __m128i b);
PMULUDQ __m64 _mm_mul_su32 (__m64 a, __m64 b)
(V)PMULUDQ __m128i _mm_mul_epu32 ( __m128i a, __m128i b)
VPMULUDQ __m256i _mm256_mul_epu32( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EB /r[1] POR mm, mm/m64	A	V/V	MMX	Bitwise OR of mm/m64 and mm.
66 0F EB /r POR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise OR of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG EB /r VPOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.
VEX.256.66.0F.WIG EB /r VPOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise OR of ymm2/m256 and ymm3.
EVEX.128.66.0F.W0 EB /r VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.
EVEX.256.66.0F.W0 EB /r VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.
EVEX.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EB /r VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.
EVEX.256.66.0F.W1 EB /r VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.
EVEX.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1 at 32/64-bit granularity.

Operation

POR (64-bit Operand)

```
DEST := DEST OR SRC
```

POR (128-bit Legacy SSE Version)

```
DEST := DEST OR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPOR (VEX.128 Encoded Version)

```
DEST := SRC1 OR SRC2
DEST[MAXVL-1:128] := 0
```

VPOR (VEX.256 Encoded Version)

```
DEST := SRC1 OR SRC2
DEST[MAXVL-1:256] := 0
```

VPORD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPORD __m512i _mm512_or_epi32(__m512i a, __m512i b);
VPORD __m512i _mm512_mask_or_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPORD __m512i _mm512_maskz_or_epi32( __mmask16 k, __m512i a, __m512i b);
VPORD __m256i _mm256_or_epi32(__m256i a, __m256i b);
VPORD __m256i _mm256_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b,);
VPORD __m256i _mm256_maskz_or_epi32( __mmask8 k, __m256i a, __m256i b);
VPORD __m128i _mm_or_epi32(__m128i a, __m128i b);
VPORD __m128i _mm_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i _mm_maskz_or_epi32( __mmask8 k, __m128i a, __m128i b);
VPORQ __m512i _mm512_or_epi64(__m512i a, __m512i b);
VPORQ __m512i _mm512_mask_or_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPORQ __m512i _mm512_maskz_or_epi64(__mmask8 k, __m512i a, __m512i b);
VPORQ __m256i _mm256_or_epi64(__m256i a, int imm);
VPORQ __m256i _mm256_mask_or_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORQ __m256i _mm256_maskz_or_epi64( __mmask8 k, __m256i a, __m256i b);
VPORQ __m128i _mm_or_epi64(__m128i a, __m128i b);
VPORQ __m128i _mm_mask_or_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORQ __m128i _mm_maskz_or_epi64( __mmask8 k, __m128i a, __m128i b);
POR __m64 _mm_or_si64(__m64 m1, __m64 m2)
(V)POR __m128i _mm_or_si128(__m128i m1, __m128i m2)
VPOR __m256i _mm256_or_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F6 /r[1] PSADBW mm1, mm2/m64	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from mm2 /m64 and mm1; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW xmm1, xmm2/m128	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2 /m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; then each consecutive 8 differences are summed separately to produce two unsigned word integer results.
EVEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Computes the absolute differences of the packed unsigned byte integers from zmm3 /m512 and zmm2; then each consecutive 8 differences are summed separately to produce eight unsigned word integer results.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	N/A

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

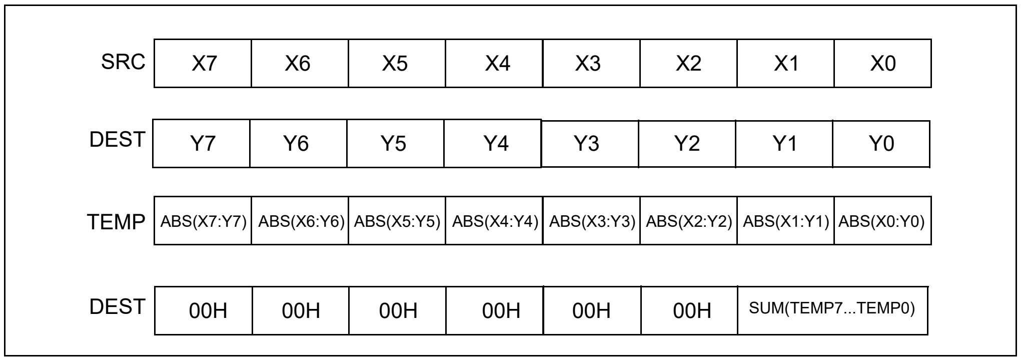


Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands

Operation

VPSADBW (EVEX Encoded Versions)

```
VL = 128, 256, 512
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 1 through 15 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H

IF VL >= 256
    (* Repeat operation for bytes 16 through 31*)
    TEMP31 := ABS(SRC1[255:248] - SRC2[255:248])
    DEST[143:128] := SUM(TEMP16:TEMP23)
    DEST[191:144] := 000000000000H
    DEST[207:192] := SUM(TEMP24:TEMP31)
    DEST[223:208] := 00000000000H
FI;
IF VL >= 512
(* Repeat operation for bytes 32 through 63*)
    TEMP63 := ABS(SRC1[511:504] - SRC2[511:504])
    DEST[271:256] := SUM(TEMP0:TEMP7)
    DEST[319:272] := 000000000000H
    DEST[335:320] := SUM(TEMP8:TEMP15)
    DEST[383:336] := 00000000000H
    DEST[399:384] := SUM(TEMP16:TEMP23)
    DEST[447:400] := 000000000000H
    DEST[463:448] := SUM(TEMP24:TEMP31)
    DEST[511:464] := 00000000000H
FI;
DEST[MAXVL-1:VL] := 0
```

VPSADBW (VEX.256 Encoded Version)

```
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 := ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H
DEST[143:128] := SUM(TEMP16:TEMP23)
DEST[191:144] := 000000000000H
DEST[207:192] := SUM(TEMP24:TEMP31)
DEST[223:208] := 00000000000H
DEST[MAXVL-1:256] := 0
```

VPSADBW (VEX.128 Encoded Version)
```
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000H
DEST[MAXVL-1:128] := 0
```

PSADBW (128-bit Legacy SSE Version)
```
TEMP0 := ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000
DEST[MAXVL-1:128] (Unmodified)
```

PSADBW (64-bit Operand)
```
TEMP0 := ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 6 *)
TEMP7 := ABS(DEST[63:56] - SRC[63:56])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSADBW __m512i _mm512_sad_epu8( __m512i a, __m512i b)
PSADBW __m64 _mm_sad_pu8(__m64 a,__m64 b)
(V)PSADBW __m128i _mm_sad_epu8(__m128i a, __m128i b)
VPSADBW __m256i _mm256_sad_epu8( __m256i a, __m256i b)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSHUFB—Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 00 /r[1] PSHUFB mm1, mm2/m64	A	V/V	SSSE3	Shuffle bytes in mm1 according to contents of mm2/m64.
66 0F 38 00 /r PSHUFB xmm1, xmm2/m128	A	V/V	SSSE3	Shuffle bytes in xmm1 according to contents of xmm2/m128.
VEX.128.66.0F38.WIG 00 /r VPSHUFB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Shuffle bytes in xmm2 according to contents of xmm3/m128.
VEX.256.66.0F38.WIG 00 /r VPSHUFB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Shuffle bytes in ymm2 according to contents of ymm3/m256.
EVEX.128.66.0F38.WIG 00 /r VPSHUFB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle bytes in xmm2 according to contents of xmm3/m128 under write mask k1.
EVEX.256.66.0F38.WIG 00 /r VPSHUFB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle bytes in ymm2 according to contents of ymm3/m256 under write mask k1.
EVEX.512.66.0F38.WIG 00 /r VPSHUFB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Shuffle bytes in zmm2 according to contents of zmm3/m512 under write mask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask. The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

Operation

PSHUFB (With 64-bit Operands)

```
TEMP := DEST
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] := 0;
    else
        index[2..0] := SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] := TEMP[(index*8+7)..(index*8+0)];
    endif;
}
PSHUFB (with 128 bit operands)
TEMP := DEST
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] := 0;
     else
        index[3..0] := SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] := TEMP[(index*8+7)..(index*8+0)];
    endif
}
```

VPSHUFB (VEX.128 Encoded Version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1) then
        DEST[(i*8)+7..(i*8)+0] := 0;
        else
        index[3..0] := SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] := SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[MAXVL-1:128] := 0
```

VPSHUFB (VEX.256 Encoded Version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] := 0;
        else
        index[3..0] := SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] := SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7..(i*8)+0] := 0;
        else
        index[3..0] := SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7..(i*8)+0] := SRC1[128 + (index*8+7)..(index*8+0)];
    endif
}
```

VPSHUFB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
jmask := (KL-1) & ~0xF                          // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1                               // dest
    IF kl[ i ] or no_masking
        index := src.byte[ j ];
        IF index & 0x80
            Dest.byte[ j ] := 0;
        ELSE
            index := (index & 0xF) + (j & jmask);       // 16-element in-lane lookup
            Dest.byte[ j ] := src.byte[ index ];
    ELSE if zeroing
        Dest.byte[ j ] := 0;
DEST[MAXVL-1:VL] := 0;
```

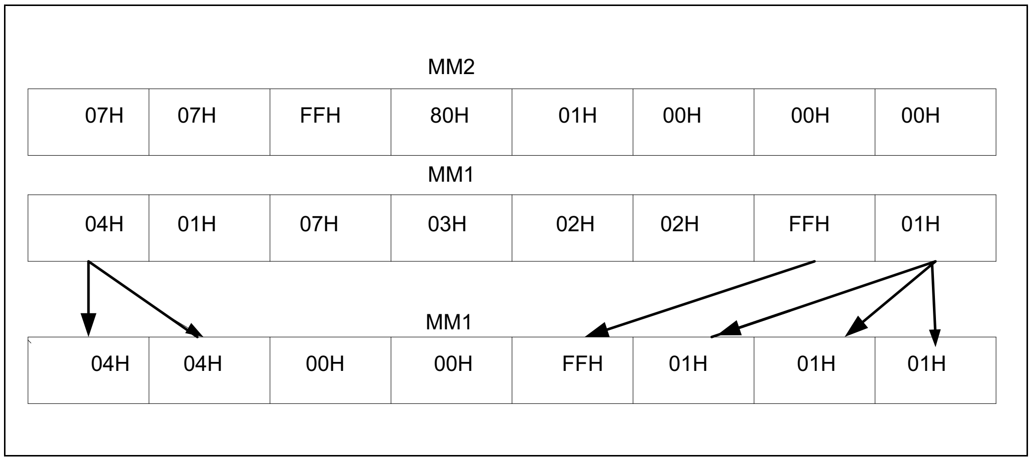


Figure 4-15. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB: __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB:__m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.66.0F.WIG 70 /r ib VPSHUFD xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.66.0F.WIG 70 /r ib VPSHUFD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the doublewords in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.66.0F.W0 70 /r ib VPSHUFD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 70 /r ib VPSHUFD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 70 /r ib VPSHUFD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	B	V/V	AVX512F OR AVX10.1	Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

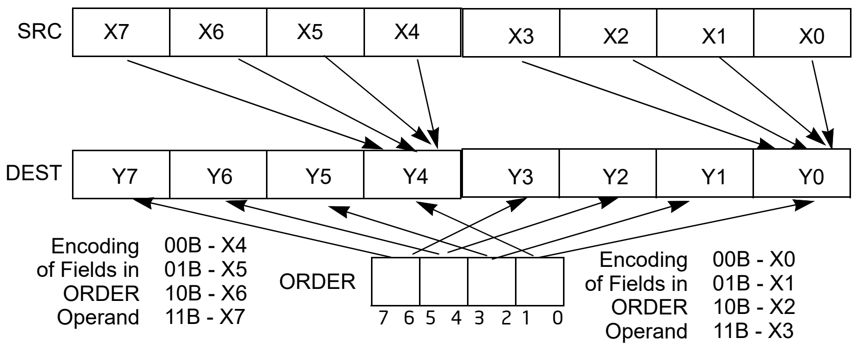


Figure 4-16. 256-bit VPSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

Note: EVEX.vvvv and VEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFD (128-bit Legacy SSE Version)

```
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFD (VEX.128 Encoded Version)

```
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] := 0
```

VPSHUFD (VEX.256 Encoded Version)

```
DEST[31:0] := (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] := (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] := (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] := (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] := (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] := (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] := (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

VPSHUFD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+31:i] := SRC[31:0]
        ELSE TMP_SRC[i+31:i] := SRC[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] := (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 256
    TMP_DEST[159:128] := (TMP_SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 512
    TMP_DEST[287:256] := (TMP_SRC[383:256] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[319:288] := (TMP_SRC[383:256] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[351:320] := (TMP_SRC[383:256] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[383:352] := (TMP_SRC[383:256] >> (ORDER[7:6] * 32))[31:0];
    TMP_DEST[415:384] := (TMP_SRC[511:384] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[447:416] := (TMP_SRC[511:384] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[479:448] := (TMP_SRC[511:384] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[511:480] := (TMP_SRC[511:384] >> (ORDER[7:6] * 32))[31:0];
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, int n );
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n );
VPSHUFD __m512i _mm512_maskz_shuffle_epi32( __mmask16 k, __m512i a, int n );
VPSHUFD __m256i _mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n );
VPSHUFD __m256i _mm256_maskz_shuffle_epi32( __mmask8 k, __m256i a, int n );
VPSHUFD __m128i _mm_mask_shuffle_epi32(__m128i s, __mmask8 k, __m128i a, int n );
VPSHUFD __m128i _mm_maskz_shuffle_epi32( __mmask8 k, __m128i a, int n );
(V)PSHUFD __m128i _mm_shuffle_epi32(__m128i a, int n)
VPSHUFD __m256i _mm256_shuffle_epi32(__m256i a, const int n)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.

PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F3.0F.WIG 70 /r ib VPSHUFHW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW OR AVX10.1	Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the write-mask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFHW (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC[63:0]
DEST[79:64] := (SRC >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFHW (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] := 0
```

VPSHUFHW (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] := SRC1[191:128]
DEST[207192] := (SRC1 >> (imm[1:0] *16))[207:192]
DEST[223:208] := (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] := (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] := (SRC1 >> (imm[7:6] * 16))[207:192]
DEST[MAXVL-1:256] := 0
```

VPSHUFHW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[79:64] := (SRC1 >> (imm[1:0] *16))[79:64]
    TMP_DEST[95:80] := (SRC1 >> (imm[3:2] * 16))[79:64]
    TMP_DEST[111:96] := (SRC1 >> (imm[5:4] * 16))[79:64]
    TMP_DEST[127:112] := (SRC1 >> (imm[7:6] * 16))[79:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[207:192] := (SRC1 >> (imm[1:0] *16))[207:192]
    TMP_DEST[223:208] := (SRC1 >> (imm[3:2] * 16))[207:192]
    TMP_DEST[239:224] := (SRC1 >> (imm[5:4] * 16))[207:192]
    TMP_DEST[255:240] := (SRC1 >> (imm[7:6] * 16))[207:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[335:320] := (SRC1 >> (imm[1:0] *16))[335:320]
    TMP_DEST[351:336] := (SRC1 >> (imm[3:2] * 16))[335:320]
    TMP_DEST[367:352] := (SRC1 >> (imm[5:4] * 16))[335:320]
    TMP_DEST[383:368] := (SRC1 >> (imm[7:6] * 16))[335:320]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[463:448] := (SRC1 >> (imm[1:0] *16))[463:448]
    TMP_DEST[479:464] := (SRC1 >> (imm[3:2] * 16))[463:448]
    TMP_DEST[495:480] := (SRC1 >> (imm[5:4] * 16))[463:448]
    TMP_DEST[511:496] := (SRC1 >> (imm[7:6] * 16))[463:448]
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHUFHW __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFHW __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n );
VPSHUFHW __m512i _mm512_maskz_shufflehi_epi16( __mmask16 k, __m512i a, int n );
VPSHUFHW __m256i _mm256_mask_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n );
VPSHUFHW __m256i _mm256_maskz_shufflehi_epi16( __mmask8 k, __m256i a, int n );
VPSHUFHW __m128i _mm_mask_shufflehi_epi16(__m128i s, __mmask8 k, __m128i a, int n );
VPSHUFHW __m128i _mm_maskz_shufflehi_epi16( __mmask8 k, __m128i a, int n );
(V)PSHUFHW __m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFHW __m256i _mm256_shufflehi_epi16(__m256i a, const int n)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.
EVEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F2.0F.WIG 70 /r ib VPSHUFLW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW OR AVX10.1	Shuffle the low words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the write-mask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFLW (128-bit Legacy SSE Version)

```
DEST[15:0] := (SRC >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFLW (VEX.128 Encoded Version)

```
DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC[127:64]
DEST[MAXVL-1:128] := 0
```

VPSHUFLW (VEX.256 Encoded Version)

```
DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] := SRC1[127:64]
DEST[143:128] := (SRC1 >> (imm[1:0] *16))[143:128]
DEST[159:144] := (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] := (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] := (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0
```

VPSHUFLW (EVEX.U1.512 Encoded Version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[15:0] := (SRC1 >> (imm[1:0] *16))[15:0]
    TMP_DEST[31:16] := (SRC1 >> (imm[3:2] * 16))[15:0]
    TMP_DEST[47:32] := (SRC1 >> (imm[5:4] * 16))[15:0]
    TMP_DEST[63:48] := (SRC1 >> (imm[7:6] * 16))[15:0]
    TMP_DEST[127:64] := SRC1[127:64]
FI;
IF VL >= 256
    TMP_DEST[143:128] := (SRC1 >> (imm[1:0] *16))[143:128]
    TMP_DEST[159:144] := (SRC1 >> (imm[3:2] * 16))[143:128]
    TMP_DEST[175:160] := (SRC1 >> (imm[5:4] * 16))[143:128]
    TMP_DEST[191:176] := (SRC1 >> (imm[7:6] * 16))[143:128]
    TMP_DEST[255:192] := SRC1[255:192]
FI;
IF VL >= 512
    TMP_DEST[271:256] := (SRC1 >> (imm[1:0] *16))[271:256]
    TMP_DEST[287:272] := (SRC1 >> (imm[3:2] * 16))[271:256]
    TMP_DEST[303:288] := (SRC1 >> (imm[5:4] * 16))[271:256]
    TMP_DEST[319:304] := (SRC1 >> (imm[7:6] * 16))[271:256]
    TMP_DEST[383:320] := SRC1[383:320]
    TMP_DEST[399:384] := (SRC1 >> (imm[1:0] *16))[399:384]
    TMP_DEST[415:400] := (SRC1 >> (imm[3:2] * 16))[399:384]
    TMP_DEST[431:416] := (SRC1 >> (imm[5:4] * 16))[399:384]
    TMP_DEST[447:432] := (SRC1 >> (imm[7:6] * 16))[399:384]
    TMP_DEST[511:448] := SRC1[511:448]
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n );
VPSHUFLW __m512i _mm512_maskz_shufflelo_epi16( __mmask16 k, __m512i a, int n );
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n );
VPSHUFLW __m256i _mm256_maskz_shufflelo_epi16( __mmask8 k, __m256i a, int n );
VPSHUFLW __m128i _mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n );
VPSHUFLW __m128i _mm_maskz_shufflelo_epi16( __mmask8 k, __m128i a, int n );
(V)PSHUFLW:__m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW:__m256i _mm256_shufflelo_epi16(__m256i a, const int n)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 left by imm8 bytes while shifting in 0s.
VEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1.
VEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm2 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1,xmm2/ m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /7 ib VPSLLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW OR AVX10.1	Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
B	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Operation

VPSLLDQ (EVEX.U1.512 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[255:128] := SRC[255:128] << (TEMP * 8)
DEST[383:256] := SRC[383:256] << (TEMP * 8)
DEST[511:384] := SRC[511:384] << (TEMP * 8)
DEST[MAXVL-1:512] := 0
```

VPSLLDQ (VEX.256 and EVEX.256 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[255:128] := SRC[255:128] << (TEMP * 8)
DEST[MAXVL-1:256] := 0
```

VPSLLDQ (VEX.128 and EVEX.128 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC << (TEMP * 8)
DEST[MAXVL-1:128] := 0
```

PSLLDQ(128-bit Legacy SSE Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST << (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)PSLLDQ __m128i _mm_slli_si128 ( __m128i a, int imm)
VPSLLDQ __m256i _mm256_slli_si256 ( __m256i a, const int imm)
VPSLLDQ __m512i _mm512_bslli_epi128 ( __m512i a, const int imm)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F1 /r[1] PSLLW mm, mm/m64	A	V/V	MMX	Shift words in mm left mm/m64 while shifting in 0s.
66 0F F1 /r PSLLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 71 /6 ib PSLLW mm1, imm8	B	V/V	MMX	Shift words in mm left by imm8 while shifting in 0s.
66 0F 71 /6 ib PSLLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 left by imm8 while shifting in 0s.
NP 0F F2 /r[1] PSLLD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm left by mm/m64 while shifting in 0s.
66 0F F2 /r PSLLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 72 /6 ib[1] PSLLD mm, imm8	B	V/V	MMX	Shift doublewords in mm left by imm8 while shifting in 0s.
66 0F 72 /6 ib PSLLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 left by imm8 while shifting in 0s.
NP 0F F3 /r[1] PSLLQ mm, mm/m64	A	V/V	MMX	Shift quadword in mm left by mm/m64 while shifting in 0s.
66 0F F3 /r PSLLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 left by xmm2/m128 while shifting in 0s.
NP 0F 73 /6 ib[1] PSLLQ mm, imm8	B	V/V	MMX	Shift quadword in mm left by imm8 while shifting in 0s.
66 0F 73 /6 ib PSLLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F1 /r VPSLLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F2 /r VPSLLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 72 /6 ib VPSLLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 left by imm8 while shifting in 0s.
VEX.128.66.0F.WIG F3 /r VPSLLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 73 /6 ib VPSLLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 left by imm8 while shifting in 0s.
VEX.256.66.0F.WIG F1 /r VPSLLW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 71 /6 ib VPSLLW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 left by imm8 while shifting in 0s.

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
VEX.256.66.0F.WIG F2 /r VPSLLD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 72 /6 ib VPSLLD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 left by imm8 while shifting in 0s.
VEX.256.66.0F.WIG F3 /r VPSLLQ ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 73 /6 ib VPSLLQ ymm1, ymm2, imm8	D	V/V	AVX2	Shift quadwords in ymm2 left by imm8 while shifting in 0s.
EVEX.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG F1 /r VPSLLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /6 ib VPSLLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2/m256 left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1	Shift words in zmm2/m512 left by imm8 while shifting in 0 using writemask k1.
EVEX.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.256.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

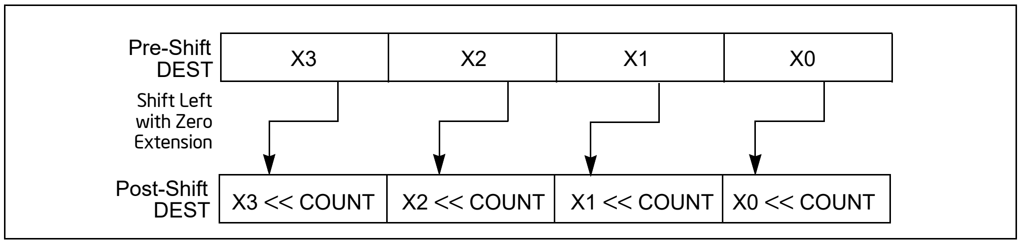


Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (With 64-bit Operand)

```
    IF (COUNT > 15)
    THEN
        DEST[64:0] := 0000000000000000H;
    ELSE
        DEST[15:0] := ZeroExtend(DEST[15:0] << COUNT);
        (* Repeat shift operation for 2nd and 3rd words *)
        DEST[63:48] := ZeroExtend(DEST[63:48] << COUNT);
    FI;
PSLLD (with 64-bit operand)
    IF (COUNT > 31)
    THEN
        DEST[64:0] := 0000000000000000H;
    ELSE
        DEST[31:0] := ZeroExtend(DEST[31:0] << COUNT);
        DEST[63:32] := ZeroExtend(DEST[63:32] << COUNT);
    FI;
```

PSLLQ (With 64-bit Operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] := 0000000000000000H;
    ELSE
```

```
            DEST := ZeroExtend(DEST << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] := ZeroExtend(SRC[127:112] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] := ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
```

```
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] := ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] := ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
    DEST[255:128] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] := ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] := ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (EVEX Versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
```

```
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*            ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLW (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLW (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLW (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLW (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLD (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLD (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLD (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLD (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLD (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLQ (EVEX Versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
```

VPSLLQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSLLQ (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSLLQ (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSLLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLQ (xmm, imm8)

```
DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
```

VPSLLQ __m512i _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_slli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW __m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW __m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW __m128i _mm_slli_epi16(__m64 m, int count)
(V)PSLLW __m128i _mm_sll_epi16(__m128i m, __m128i count)
VPSLLW __m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW __m256i _mm256_sll_epi16 (__m256i m, __m128i count)
PSLLD __m64 _mm_slli_pi32(__m64 m, int count)
PSLLD __m64 _mm_sll_pi32(__m64 m, __m64 count)
(V)PSLLD __m128i _mm_slli_epi32(__m128i m, int count)
(V)PSLLD __m128i _mm_sll_epi32(__m128i m, __m128i count)
VPSLLD __m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD __m256i _mm256_sll_epi32 (__m256i m, __m128i count)
PSLLQ __m64 _mm_slli_si64(__m64 m, int count)
PSLLQ __m64 _mm_sll_si64(__m64 m, __m64 count)
(V)PSLLQ __m128i _mm_slli_epi64(__m128i m, int count)
(V)PSLLQ __m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLQ __m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ __m256i _mm256_sll_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:

  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, “Type 4 Class Exception Conditions.”
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, “Type 7 Class Exception Conditions.”
- EVEX-encoded VPSLLW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”
- EVEX-encoded VPSLLD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-51, “Type E4 Class Exception Conditions.”

PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E1 /r[1] PSRAW mm, mm/m64	A	V/V	MMX	Shift words in mm right by mm/m64 while shifting in sign bits.
66 0F E1 /r PSRAW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by xmm2/m128 while shifting in sign bits.
NP 0F 71 /4 ib[1] PSRAW mm, imm8	B	V/V	MMX	Shift words in mm right by imm8 while shifting in sign bits
66 0F 71 /4 ib PSRAW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in sign bits
NP 0F E2 /r[1] PSRAD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm right by mm/m64 while shifting in sign bits.
66 0F E2 /r PSRAD xmm1, xmm2/m128	A	V/V	SSE2	Shift doubleword in xmm1 right by xmm2 /m128 while shifting in sign bits.
NP 0F 72 /4 ib[1] PSRAD mm, imm8	B	V/V	MMX	Shift doublewords in mm right by imm8 while shifting in sign bits.
66 0F 72 /4 ib PSRAD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.
VEX.128.66.0F.WIG E1 /r VPSRAW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.128.66.0F.WIG 71 /4 ib VPSRAW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in sign bits.
VEX.128.66.0F.WIG E2 /r VPSRAD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.128.66.0F.WIG 72 /4 ib VPSRAD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in sign bits.
VEX.256.66.0F.WIG E1 /r VPSRAW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.256.66.0F.WIG 71 /4 ib VPSRAW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in sign bits.
VEX.256.66.0F.WIG E2 /r VPSRAD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.256.66.0F.WIG 72 /4 ib VPSRAD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in sign bits.
EVEX.128.66.0F.WIG E1 /r VPSRAW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG E1 /r VPSRAW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.WIG E1 /r VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW)OR AVX10.1[2]	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

NOTES:

1. See note in Section 2.5, “Intel® AVX and Intel® SSE Instruction Exception Classification,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, and Section 24.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

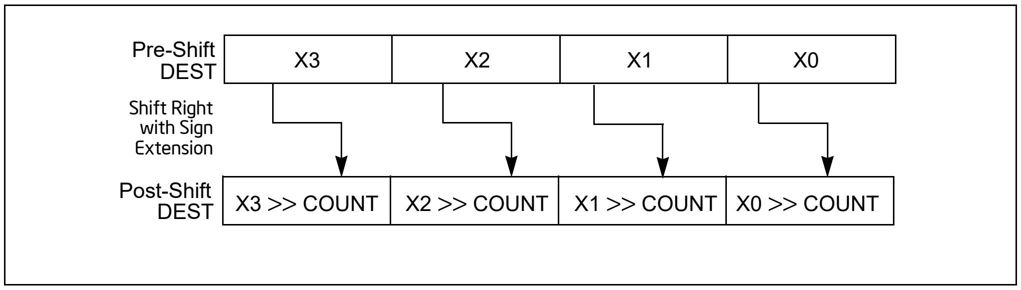


Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRAW (With 64-bit Operand)

```
    IF (COUNT > 15)
        THEN COUNT := 16;
    FI;
    DEST[15:0] := SignExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] := SignExtend(DEST[63:48] >> COUNT);

PSRAD (with 64-bit operand)
    IF (COUNT > 31)
        THEN COUNT := 32;
    FI;
    DEST[31:0] := SignExtend(DEST[31:0] >> COUNT);
    DEST[63:32] := SignExtend(DEST[63:32] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := SignBit
ELSE
    DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := SignBit
ELSE
    DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT := 16;
FI;
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] := SignExtend(SRC[255:240] >> COUNT);
```

```
ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT := 32;
FI;
DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] := SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL) ; VL: 128b, 256b or 512b
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT := 64;
FI;
DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] := SignExtend(SRC[VL-1:VL-64] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT := 16;
FI;
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] := SignExtend(SRC[127:112] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT := 32;
FI;
DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] := SignExtend(SRC[127:96] >> COUNT);
```

VPSRAW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAW (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPSRAW (ymm, imm8) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0
```

VPSRAW (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRAW (xmm, imm8) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAD (EVEX Versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
               THEN DEST[i+31:i] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
               ELSE DEST[i+31:i] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
      ELSE
            IF *merging-masking*                ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE *zeroing-masking*           ; zeroing-masking
                  DEST[i+31:i] := 0
            FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
   TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
   TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
   TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
   TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TMP_DEST[i+31:i]
      ELSE
            IF *merging-masking*                ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE *zeroing-masking*           ; zeroing-masking
                  DEST[i+31:i] := 0
            FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAD (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPSRAD (ymm, imm8) - VEX

```
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0
```

VPSRAD (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRAD (xmm, imm8) - VEX

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAD (xmm, imm8)

```
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAQ (EVEX Versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
               THEN DEST[i+63:i] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
               ELSE DEST[i+63:i] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
         ELSE
            IF *merging-masking*                  ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
               ELSE *zeroing-masking*             ; zeroing-masking
                  DEST[i+63:i] := 0
            FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRAQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[VL-1:0] := ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC1[VL-1:0], SRC2, VL)

FOR j := 0 TO 7
   i := j * 64
   IF k1[j] OR *no writemask*
         THEN DEST[i+63:i] := TMP_DEST[i+63:i]
         ELSE
            IF *merging-masking*                  ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
               ELSE *zeroing-masking*             ; zeroing-masking
```

```
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm_maskz_srai_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm_maskz_sra_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m512i _mm512_maskz_srai_epi64( __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64( __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm_maskz_srai_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i _mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m512i _mm512_maskz_sra_epi64( __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m256i _mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i _mm256_maskz_sra_epi64( __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i _mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i _mm_maskz_sra_epi64( __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m512i _mm512_maskz_srai_epi16( __mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m256i _mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i _mm256_maskz_srai_epi16( __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m128i _mm_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m128i _mm_maskz_srai_epi16( __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m512i _mm512_sra_epi16(__m512i a, __m128i cnt);
VPSRAW __m512i _mm512_mask_sra_epi16(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m512i _mm512_maskz_sra_epi16( __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m256i _mm256_mask_sra_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m256i _mm256_maskz_sra_epi16( __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m128i _mm_mask_sra_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m128i _mm_maskz_sra_epi16( __mmask8 k, __m128i a, __m128i cnt);
PSRAW __m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW __m64 _mm_sra_pi16 (__m64 m, __m64 count)
(V)PSRAW __m128i _mm_srai_epi16(__m128i m, int count)
(V)PSRAW __m128i _mm_sra_epi16(__m128i m, __m128i count)
```

```
VPSRAW __m256i _mm256_srai_epi16 (__m256i m, int count)
VPSRAW __m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD __m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD __m64 _mm_sra_pi32 (__m64 m, __m64 count)
(V)PSRAD __m128i _mm_srai_epi32 (__m128i m, int count)
(V)PSRAD __m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD __m256i _mm256_srai_epi32 (__m256i m, int count)
VPSRAD __m256i _mm256_sra_epi32 (__m256i m, __m128i count)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, "Type 4 Class Exception Conditions."
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, "Type 7 Class Exception Conditions."
- EVEX-encoded VPSRAW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
- EVEX-encoded VPSRAD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-51, "Type E4 Class Exception Conditions."

PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 right by imm8 bytes while shifting in 0s.
VEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm1 right by imm8 bytes while shifting in 0s.
EVEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2/m128, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift xmm2/m128 right by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2/m256, imm8	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift ymm2/m256 right by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /3 ib VPSRLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW OR AVX10.1	Shift zmm2/m512 right by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
B	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

VPSRLDQ (EVEX.512 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[255:128] := SRC[255:128] >> (TEMP * 8)
DEST[383:256] := SRC[383:256] >> (TEMP * 8)
DEST[511:384] := SRC[511:384] >> (TEMP * 8)
DEST[MAXVL-1:512] := 0;
```

VPSRLDQ (VEX.256 and EVEX.256 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[255:128] := SRC[255:128] >> (TEMP * 8)
DEST[MAXVL-1:256] := 0;
```

VPSRLDQ (VEX.128 and EVEX.128 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC >> (TEMP * 8)
DEST[MAXVL-1:128] := 0;
```

PSRLDQ (128-bit Legacy SSE Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST >> (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
(V)PSRLDQ __m128i _mm_srli_si128 ( __m128i a, int imm)
VPSRLDQ __m256i _mm256_bsrli_epi128 ( __m256i, const int)
VPSRLDQ __m512i _mm512_bsrli_epi128 ( __m512i, int)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D1 /r[1] PSRLW mm, mm/m64	A	V/V	MMX	Shift words in mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D1 /r PSRLW xmm1, xmm2/m128	A	V/V	SSE2	Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
NP 0F 71 /2 ib[1] PSRLW mm, imm8	B	V/V	MMX	Shift words in mm right by imm8 while shifting in 0s.
66 0F 71 /2 ib PSRLW xmm1, imm8	B	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in 0s.
NP 0F D2 /r[1] PSRLD mm, mm/m64	A	V/V	MMX	Shift doublewords in mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D2 /r PSRLD xmm1, xmm2/m128	A	V/V	SSE2	Shift doublewords in xmm1 right by amount specified in xmm2 /m128 while shifting in 0s.
NP 0F 72 /2 ib[1] PSRLD mm, imm8	B	V/V	MMX	Shift doublewords in mm right by imm8 while shifting in 0s.
66 0F 72 /2 ib PSRLD xmm1, imm8	B	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in 0s.
NP 0F D3 /r[1] PSRLQ mm, mm/m64	A	V/V	MMX	Shift mm right by amount specified in mm/m64 while shifting in 0s.
66 0F D3 /r PSRLQ xmm1, xmm2/m128	A	V/V	SSE2	Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.
NP 0F 73 /2 ib[1] PSRLQ mm, imm8	B	V/V	MMX	Shift mm right by imm8 while shifting in 0s.
66 0F 73 /2 ib PSRLQ xmm1, imm8	B	V/V	SSE2	Shift quadwords in xmm1 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D1 /r VPSRLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D2 /r VPSRLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 72 /2 ib VPSRLD xmm1, xmm2, imm8	D	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in 0s.
VEX.128.66.0F.WIG D3 /r VPSRLQ xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.128.66.0F.WIG 73 /2 ib VPSRLQ xmm1, xmm2, imm8	D	V/V	AVX	Shift quadwords in xmm2 right by imm8 while shifting in 0s.
VEX.256.66.0F.WIG D1 /r VPSRLW ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1, ymm2, imm8	D	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in 0s.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F.WIG D2 /r VPSRLD ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 72 /2 ib VPSRLD ymm1, ymm2, imm8	D	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in 0s.
VEX.256.66.0F.WIG D3 /r VPSRLQ ymm1, ymm2, xmm3/m128	C	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.256.66.0F.WIG 73 /2 ib VPSRLQ ymm1, ymm2, imm8	D	V/V	AVX2	Shift quadwords in ymm2 right by imm8 while shifting in 0s.
EVEX.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW OR AVX10.1	Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:r/m (r, w)	imm8	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	VEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
F	Full	EVEX.vvvv (w)	ModRM:r/m (r)	imm8	N/A
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

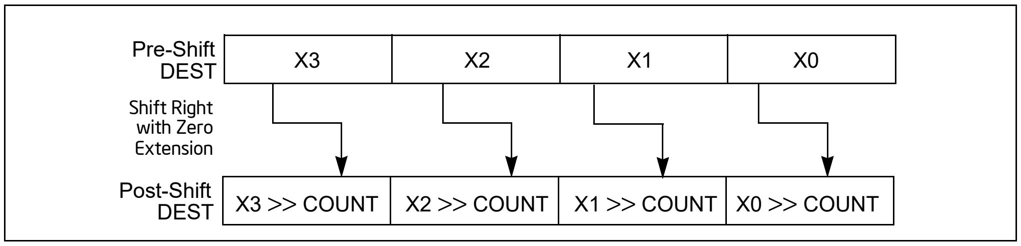


Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (With 64-bit Operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] := 0000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] := ZeroExtend(DEST[63:48] >> COUNT);
FI;
```

PSRLD (With 64-bit Operand)

```
    IF (COUNT > 31)
    THEN
        DEST[64:0] := 0000000000000000H
    ELSE
        DEST[31:0] := ZeroExtend(DEST[31:0] >> COUNT);
        DEST[63:32] := ZeroExtend(DEST[63:32] >> COUNT);
    FI;
```

PSRLQ (With 64-bit Operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] := 0000000000000000H
    ELSE
        DEST := ZeroExtend(DEST >> COUNT);
    FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] := 0
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] := ZeroExtend(SRC[255:240] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] := ZeroExtend(SRC[127:112] >> COUNT);
FI;
```

```
LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] := ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] := ZeroExtend(SRC[127:96] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] := ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] := ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
FI;
```

VPSRLW (EVEX Versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLW (EVEX Versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLW (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLW (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLD (EVEX Versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLD (EVEX Versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] := LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] := LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
```

```
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLD (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLD (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLD (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLD (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLQ (EVEX Versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLQ (EVEX Versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;
```

VPSRLQ (ymm, imm8) - VEX.256 Encoding

```
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
```

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPSRLQ (xmm, imm8) - VEX.128 Encoding

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
```

PSRLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLQ (xmm, imm8)

```
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
```

VPSRLD __m256i _mm256_maskz_srli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_srl_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srl_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srli_epi64(__m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_srli_epi16(__m512i a, unsigned int imm);
VPSRLW __m512i _mm512_mask_srli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i _mm256_mask_srli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i _mm_mask_srli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i _mm512_srl_epi16(__m512i a, __m128i cnt);
VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i _mm256_mask_srl_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i _mm256_maskz_srl_epi16(__mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i _mm_mask_srl_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRLW __m64 _mm_srli_pi16(__m64 m, int count)
PSRLW __m64 _mm_srl_pi16 (__m64 m, __m64 count)
(V)PSRLW __m128i _mm_srli_epi16 (__m128i m, int count)
(V)PSRLW __m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW __m256i _mm256_srli_epi16 (__m256i m, int count)
VPSRLW __m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD __m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD __m64 _mm_srl_pi32 (__m64 m, __m64 count)
(V)PSRLD __m128i _mm_srli_epi32 (__m128i m, int count)
(V)PSRLD __m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD __m256i _mm256_srli_epi32 (__m256i m, int count)
VPSRLD __m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ __m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ __m64 _mm_srl_si64 (__m64 m, __m64 count)
(V)PSRLQ __m128i _mm_srli_epi64 (__m128i m, int count)
(V)PSRLQ __m128i _mm_srl_epi64 (__m128i m, __m128i count)

VPSRLQ __m256i _mm256_srli_epi64 (__m256i m, int count)
VPSRLQ __m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, "Type 4 Class Exception Conditions."
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, "Type 7 Class Exception Conditions."
- EVEX-encoded VPSRLW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
- EVEX-encoded VPSRLD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-51, "Type E4 Class Exception Conditions."

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F8 /r[1] PSUBB mm, mm/m64	A	V/V	MMX	Subtract packed byte integers in mm/m64 from packed byte integers in mm.
66 0F F8 /r PSUBB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed byte integers in xmm2/m128 from packed byte integers in xmm1.
NP 0F F9 /r[1] PSUBW mm, mm/m64	A	V/V	MMX	Subtract packed word integers in mm/m64 from packed word integers in mm.
66 0F F9 /r PSUBW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed word integers in xmm2/m128 from packed word integers in xmm1.
NP 0F FA /r[1] PSUBD mm, mm/m64	A	V/V	MMX	Subtract packed doubleword integers in mm/m64 from packed doubleword integers in mm.
66 0F FA /r PSUBD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed doubleword integers in xmm2/mem128 from packed doubleword integers in xmm1.
VEX.128.66.0F.WIG F8 /r VPSUBB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed byte integers in xmm3/m128 from xmm2.
VEX.128.66.0F.WIG F9 /r VPSUBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed word integers in xmm3/m128 from xmm2.
VEX.128.66.0F.WIG FA /r VPSUBD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed doubleword integers in xmm3/m128 from xmm2.
VEX.256.66.0F.WIG F8 /r VPSUBB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed byte integers in ymm3/m256 from ymm2.
VEX.256.66.0F.WIG F9 /r VPSUBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed word integers in ymm3/m256 from ymm2.
VEX.256.66.0F.WIG FA /r VPSUBD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed doubleword integers in ymm3/m256 from ymm2.
EVEX.128.66.0F.WIG F8 /r VPSUBB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F8 /r VPSUBB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F8 /r VPSUBB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG F9 /r VPSUBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F9 /r VPSUBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F9 /r VPSUBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBB (With 64-bit Operands)

```
    DEST[7:0] := DEST[7:0] – SRC[7:0];
    (* Repeat subtract operation for 2nd through 7th byte *)
    DEST[63:56] := DEST[63:56] – SRC[63:56];
```

PSUBW (With 64-bit Operands)

```
    DEST[15:0] := DEST[15:0] – SRC[15:0];
    (* Repeat subtract operation for 2nd and 3rd word *)
    DEST[63:48] := DEST[63:48] – SRC[63:48];
```

PSUBD (With 64-bit Operands)

```
    DEST[31:0] := DEST[31:0] – SRC[31:0];
    DEST[63:32] := DEST[63:32] – SRC[63:32];
```

PSUBD (With 128-bit Operands)

```
    DEST[31:0] := DEST[31:0] – SRC[31:0];
    (* Repeat subtract operation for 2nd and 3rd doubleword *)
    DEST[127:96] := DEST[127:96] – SRC[127:96];
```

VPSUBB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC1[i+7:i] - SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC1[i+15:i] - SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] - SRC2[31:0]
                ELSE DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBB (VEX.256 Encoded Version)

```
DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[15:8] := SRC1[15:8]-SRC2[15:8]
DEST[23:16] := SRC1[23:16]-SRC2[23:16]
DEST[31:24] := SRC1[31:24]-SRC2[31:24]
DEST[39:32] := SRC1[39:32]-SRC2[39:32]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[55:48] := SRC1[55:48]-SRC2[55:48]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[79:72] := SRC1[79:72]-SRC2[79:72]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[95:88] := SRC1[95:88]-SRC2[95:88]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
DEST[111:104] := SRC1[111:104]-SRC2[111:104]
DEST[119:112] := SRC1[119:112]-SRC2[119:112]
DEST[127:120] := SRC1[127:120]-SRC2[127:120]
DEST[135:128] := SRC1[135:128]-SRC2[135:128]
DEST[143:136] := SRC1[143:136]-SRC2[143:136]
```

```
DEST[151:144] := SRC1[151:144]-SRC2[151:144]
DEST[159:152] := SRC1[159:152]-SRC2[159:152]
DEST[167:160] := SRC1[167:160]-SRC2[167:160]
DEST[175:168] := SRC1[175:168]-SRC2[175:168]
DEST[183:176] := SRC1[183:176]-SRC2[183:176]
DEST[191:184] := SRC1[191:184]-SRC2[191:184]
DEST[199:192] := SRC1[199:192]-SRC2[199:192]
DEST[207:200] := SRC1[207:200]-SRC2[207:200]
DEST[215:208] := SRC1[215:208]-SRC2[215:208]
DEST[223:216] := SRC1[223:216]-SRC2[223:216]
DEST[231:224] := SRC1[231:224]-SRC2[231:224]
DEST[239:232] := SRC1[239:232]-SRC2[239:232]
DEST[247:240] := SRC1[247:240]-SRC2[247:240]
DEST[255:248] := SRC1[255:248]-SRC2[255:248]
DEST[MAXVL-1:256] := 0
```

VPSUBB (VEX.128 Encoded Version)

```
DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[15:8] := SRC1[15:8]-SRC2[15:8]
DEST[23:16] := SRC1[23:16]-SRC2[23:16]
DEST[31:24] := SRC1[31:24]-SRC2[31:24]
DEST[39:32] := SRC1[39:32]-SRC2[39:32]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[55:48] := SRC1[55:48]-SRC2[55:48]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[79:72] := SRC1[79:72]-SRC2[79:72]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[95:88] := SRC1[95:88]-SRC2[95:88]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
DEST[111:104] := SRC1[111:104]-SRC2[111:104]
DEST[119:112] := SRC1[119:112]-SRC2[119:112]
DEST[127:120] := SRC1[127:120]-SRC2[127:120]
DEST[MAXVL-1:128] := 0
```

PSUBB (128-bit Legacy SSE Version)

```
DEST[7:0] := DEST[7:0]-SRC[7:0]
DEST[15:8] := DEST[15:8]-SRC[15:8]
DEST[23:16] := DEST[23:16]-SRC[23:16]
DEST[31:24] := DEST[31:24]-SRC[31:24]
DEST[39:32] := DEST[39:32]-SRC[39:32]
DEST[47:40] := DEST[47:40]-SRC[47:40]
DEST[55:48] := DEST[55:48]-SRC[55:48]
DEST[63:56] := DEST[63:56]-SRC[63:56]
DEST[71:64] := DEST[71:64]-SRC[71:64]
DEST[79:72] := DEST[79:72]-SRC[79:72]
DEST[87:80] := DEST[87:80]-SRC[87:80]
DEST[95:88] := DEST[95:88]-SRC[95:88]
DEST[103:96] := DEST[103:96]-SRC[103:96]
DEST[111:104] := DEST[111:104]-SRC[111:104]
DEST[119:112] := DEST[119:112]-SRC[119:112]
DEST[127:120] := DEST[127:120]-SRC[127:120]
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBW (VEX.256 Encoded Version)
```
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[31:16] := SRC1[31:16]-SRC2[31:16]
DEST[47:32] := SRC1[47:32]-SRC2[47:32]
DEST[63:48] := SRC1[63:48]-SRC2[63:48]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[95:80] := SRC1[95:80]-SRC2[95:80]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[127:112] := SRC1[127:112]-SRC2[127:112]
DEST[143:128] := SRC1[143:128]-SRC2[143:128]
DEST[159:144] := SRC1[159:144]-SRC2[159:144]
DEST[175:160] := SRC1[175:160]-SRC2[175:160]
DEST[191:176] := SRC1[191:176]-SRC2[191:176]
DEST[207:192] := SRC1207:192]-SRC2[207:192]
DEST[223:208] := SRC1[223:208]-SRC2[223:208]
DEST[239:224] := SRC1[239:224]-SRC2[239:224]
DEST[255:240] := SRC1[255:240]-SRC2[255:240]
DEST[MAXVL-1:256] := 0
```

VPSUBW (VEX.128 Encoded Version)
```
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[31:16] := SRC1[31:16]-SRC2[31:16]
DEST[47:32] := SRC1[47:32]-SRC2[47:32]
DEST[63:48] := SRC1[63:48]-SRC2[63:48]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[95:80] := SRC1[95:80]-SRC2[95:80]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[127:112] := SRC1[127:112]-SRC2[127:112]
DEST[MAXVL-1:128] := 0
```

PSUBW (128-bit Legacy SSE Version)
```
DEST[15:0] := DEST[15:0]-SRC[15:0]
DEST[31:16] := DEST[31:16]-SRC[31:16]
DEST[47:32] := DEST[47:32]-SRC[47:32]
DEST[63:48] := DEST[63:48]-SRC[63:48]
DEST[79:64] := DEST[79:64]-SRC[79:64]
DEST[95:80] := DEST[95:80]-SRC[95:80]
DEST[111:96] := DEST[111:96]-SRC[111:96]
DEST[127:112] := DEST[127:112]-SRC[127:112]
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBD (VEX.256 Encoded Version)
```
DEST[31:0] := SRC1[31:0]-SRC2[31:0]
DEST[63:32] := SRC1[63:32]-SRC2[63:32]
DEST[95:64] := SRC1[95:64]-SRC2[95:64]
DEST[127:96] := SRC1[127:96]-SRC2[127:96]
DEST[159:128] := SRC1[159:128]-SRC2[159:128]
DEST[191:160] := SRC1[191:160]-SRC2[191:160]
DEST[223:192] := SRC1[223:192]-SRC2[223:192]
DEST[255:224] := SRC1[255:224]-SRC2[255:224]
DEST[MAXVL-1:256] := 0
```

VPSUBD (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0]-SRC2[31:0]
DEST[63:32] := SRC1[63:32]-SRC2[63:32]
DEST[95:64] := SRC1[95:64]-SRC2[95:64]
DEST[127:96] := SRC1[127:96]-SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

PSUBD (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0]-SRC[31:0]
DEST[63:32] := DEST[63:32]-SRC[63:32]
DEST[95:64] := DEST[95:64]-SRC[95:64]
DEST[127:96] := DEST[127:96]-SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8( __mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi8( __mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8( __mmask16 k, __m128i a, __m128i b);
VPSUBW __m512i _mm512_sub_epi16(__m512i a, __m512i b);
VPSUBW __m512i _mm512_mask_sub_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBW __m512i _mm512_maskz_sub_epi16( __mmask32 k, __m512i a, __m512i b);
VPSUBW __m256i _mm256_mask_sub_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBW __m256i _mm256_maskz_sub_epi16( __mmask16 k, __m256i a, __m256i b);
VPSUBW __m128i _mm_mask_sub_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBW __m128i _mm_maskz_sub_epi16( __mmask8 k, __m128i a, __m128i b);
VPSUBD __m512i _mm512_sub_epi32(__m512i a, __m512i b);
VPSUBD __m512i _mm512_mask_sub_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPSUBD __m512i _mm512_maskz_sub_epi32( __mmask16 k, __m512i a, __m512i b);
VPSUBD __m256i _mm256_mask_sub_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBD __m256i _mm256_maskz_sub_epi32( __mmask8 k, __m256i a, __m256i b);
VPSUBD __m128i _mm_mask_sub_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBD __m128i _mm_maskz_sub_epi32( __mmask8 k, __m128i a, __m128i b);
PSUBB __m64 _mm_sub_pi8(__m64 m1, __m64 m2)
(V)PSUBB __m128i _mm_sub_epi8 ( __m128i a, __m128i b)
VPSUBB __m256i _mm256_sub_epi8 ( __m256i a, __m256i b)
PSUBW __m64 _mm_sub_pi16(__m64 m1, __m64 m2)
(V)PSUBW __m128i _mm_sub_epi16 ( __m128i a, __m128i b)
VPSUBW __m256i _mm256_sub_epi16 ( __m256i a, __m256i b)
PSUBD __m64 _mm_sub_pi32(__m64 m1, __m64 m2)
(V)PSUBD __m128i _mm_sub_epi32 ( __m128i a, __m128i b)
VPSUBD __m256i _mm256_sub_epi32 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPSUBD, see Table 2-51, “Type E4 Class Exception Conditions.”

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FB /r[1] PSUBQ mm1, mm2/m64	A	V/V	SSE2	Subtract quadword integer in mm1 from mm2 /m64.
66 0F FB /r PSUBQ xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed quadword integers in xmm1 from xmm2 /m128.
VEX.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.
VEX.256.66.0F.WIG FB /r VPSUBQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed quadword integers in ymm3/m256 from ymm2.
EVEX.128.66.0F.W1 FB /r VPSUBQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W1 FB /r VPSUBQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBQ (With 64-Bit Operands)

```
    DEST[63:0] := DEST[63:0] − SRC[63:0];
```

PSUBQ (With 128-Bit Operands)

```
    DEST[63:0] := DEST[63:0] − SRC[63:0];
    DEST[127:64] := DEST[127:64] − SRC[127:64];
```

VPSUBQ (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]-SRC2[63:0]
DEST[127:64] := SRC1[127:64]-SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

VPSUBQ (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]-SRC2[63:0]
DEST[127:64] := SRC1[127:64]-SRC2[127:64]
DEST[191:128] := SRC1[191:128]-SRC2[191:128]
DEST[255:192] := SRC1[255:192]-SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VPSUBQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[i+63:i] := SRC1[i+63:i] - SRC2[63:0]
                    ELSE DEST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i]
                FI;
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64( __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64( __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64( __mmask8 k, __m128i a, __m128i b);
PSUBQ __m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPSUBQ, see Table 2-51, “Type E4 Class Exception Conditions.”

PSUBSB/PSUBSW—Subtract Packed Signed Integers With Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E8 /r[1] PSUBSB mm, mm/m64	A	V/V	MMX	Subtract signed packed bytes in mm/m64 from signed packed bytes in mm and saturate results.
66 0F E8 /r PSUBSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.
NP 0F E9 /r[1] PSUBSW mm, mm/m64	A	V/V	MMX	Subtract signed packed words in mm/m64 from signed packed words in mm and saturate results.
66 0F E9 /r PSUBSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.
VEX.128.66.0F.WIG E8 /r VPSUBSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results.
VEX.128.66.0F.WIG E9 /r VPSUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results.
VEX.256.66.0F.WIG E8 /r VPSUBSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results.
VEX.256.66.0F.WIG E9 /r VPSUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results.
EVEX.128.66.0F.WIG E8 /r VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E8 /r VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG E8 /r VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG E9 /r VPSUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E9 /r VPSUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBSB (With 64-bit Operands)

```
    DEST[7:0] := SaturateToSignedByte (DEST[7:0] – SRC (7:0]);
    (* Repeat subtract operation for 2nd through 7th bytes *)
    DEST[63:56] := SaturateToSignedByte (DEST[63:56] – SRC[63:56] );
```

PSUBSW (With 64-bit Operands)

```
    DEST[15:0] := SaturateToSignedWord (DEST[15:0] − SRC[15:0] );
    (* Repeat subtract operation for 2nd and 7th words *)
    DEST[63:48] := SaturateToSignedWord (DEST[63:48] − SRC[63:48] );
```

VPSUBSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPSUBSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSUBSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] := SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] := 0;
```

VPSUBSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] := 0;
```

PSUBSB (128-bit Legacy SSE Version)

```
DEST[7:0] := SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified);
```

VPSUBSW (VEX.256 Encoded Version)
```
DEST[15:0] := SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] := SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] := 0;
```

VPSUBSW (VEX.128 Encoded Version)
```
DEST[15:0] := SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] := 0;
```

PSUBSW (128-bit Legacy SSE Version)
```
DEST[15:0] := SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified);
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8( __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8( __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm_mask_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm_maskz_subs_epi8( __mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16( __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_mask_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16( __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm_mask_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm_maskz_subs_epi16( __mmask8 k, __m128i a, __m128i b);
PSUBSB __m64 _mm_subs_pi8(__m64 m1, __m64 m2)
(V)PSUBSB __m128i _mm_subs_epi8(__m128i m1, __m128i m2)
VPSUBSB __m256i _mm256_subs_epi8(__m256i m1, __m256i m2)
PSUBSW __m64 _mm_subs_pi16(__m64 m1, __m64 m2)
(V)PSUBSW __m128i _mm_subs_epi16(__m128i m1, __m128i m2)
VPSUBSW __m256i _mm256_subs_epi16(__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers With Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D8 /r[1] PSUBUSB mm, mm/m64	A	V/V	MMX	Subtract unsigned packed bytes in mm/m64 from unsigned packed bytes in mm and saturate result.
66 0F D8 /r PSUBUSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned byte integers in xmm2/m128 from packed unsigned byte integers in xmm1 and saturate result.
NP 0F D9 /r[1] PSUBUSW mm, mm/m64	A	V/V	MMX	Subtract unsigned packed words in mm/m64 from unsigned packed words in mm and saturate result.
66 0F D9 /r PSUBUSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned word integers in xmm2/m128 from packed unsigned word integers in xmm1 and saturate result.
VEX.128.66.0F.WIG D8 /r VPSUBUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2 and saturate result.
VEX.128.66.0F.WIG D9 /r VPSUBUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate result.
VEX.256.66.0F.WIG D8 /r VPSUBUSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2 and saturate result.
VEX.256.66.0F.WIG D9 /r VPSUBUSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2 and saturate result.
EVEX.128.66.0F.WIG D8 /r VPSUBUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D8 /r VPSUBUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG D8 /r VPSUBUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG D9 /r VPSUBUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D9 /r VPSUBUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBUSB (With 64-bit Operands)

```
    DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] – SRC (7:0] );
    (* Repeat add operation for 2nd through 7th bytes *)
    DEST[63:56] := SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];
```

PSUBUSW (With 64-bit Operands)

```
    DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] − SRC[15:0] );
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] := SaturateToUnsignedWord (DEST[63:48] − SRC[63:48] );
```

VPSUBUSB (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSUBUSW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16;
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] := 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSUBUSB (VEX.256 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] := SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] := 0;
```

VPSUBUSB (VEX.128 Encoded Version)

```
DEST[7:0] := SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] := 0
```

PSUBUSB (128-bit Legacy SSE Version)

```
DEST[7:0] := SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] := SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBUSW (VEX.256 Encoded Version)

```
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] := SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] := 0;
```

VPSUBUSW (VEX.128 Encoded Version)

```
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] := 0
```

PSUBUSW (128-bit Legacy SSE Version)

```
DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] := SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8( __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8( __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu8( __mmask16 k, __m128i a, __m128i b);
VPSUBUSW __m512i _mm512_subs_epu16(__m512i a, __m512i b);
VPSUBUSW __m512i _mm512_mask_subs_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m512i _mm512_maskz_subs_epu16( __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m256i _mm256_mask_subs_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m256i _mm256_maskz_subs_epu16( __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m128i _mm_mask_subs_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBUSW __m128i _mm_maskz_subs_epu16( __mmask8 k, __m128i a, __m128i b);
PSUBUSB __m64 _mm_subs_pu8(__m64 m1, __m64 m2)
(V)PSUBUSB __m128i _mm_subs_epu8(__m128i m1, __m128i m2)
VPSUBUSB __m256i _mm256_subs_epu8(__m256i m1, __m256i m2)
PSUBUSW __m64 _mm_subs_pu16(__m64 m1, __m64 m2)
(V)PSUBUSW __m128i _mm_subs_epu16(__m128i m1, __m128i m2)
VPSUBUSW __m256i _mm256_subs_epu16(__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 68 /r[1] PUNPCKHBW mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order bytes from mm and mm/m64 into mm.
66 0F 68 /r PUNPCKHBW xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.
NP 0F 69 /r[1] PUNPCKHWD mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order words from mm and mm/m64 into mm.
66 0F 69 /r PUNPCKHWD xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order words from xmm1 and xmm2/m128 into xmm1.
NP 0F 6A /r[1] PUNPCKHDQ mm, mm/m64	A	V/V	MMX	Unpack and interleave high-order doublewords from mm and mm/m64 into mm.
66 0F 6A /r PUNPCKHDQ xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6D /r PUNPCKHQDQ xmm1, xmm2/m128	A	V/V	SSE2	Unpack and interleave high-order quadwords from xmm1 and xmm2/m128 into xmm1.
VEX.128.66.0F.WIG 68/r VPUNPCKHBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 69/r VPUNPCKHWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order words from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6A/r VPUNPCKHDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6D/r VPUNPCKHQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6A /r VPUNPCKHDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6D /r VPUNPCKHQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order quadword from ymm2 and ymm3/m256 into ymm1 register.
EVEX.128.66.0F.WIG 68 /r VPUNPCKHBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.WIG 69 /r VPUNPCKHWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.W0 6A /r VPUNPCKHDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask.
EVEX.128.66.0F.W1 6D /r VPUNPCKHQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

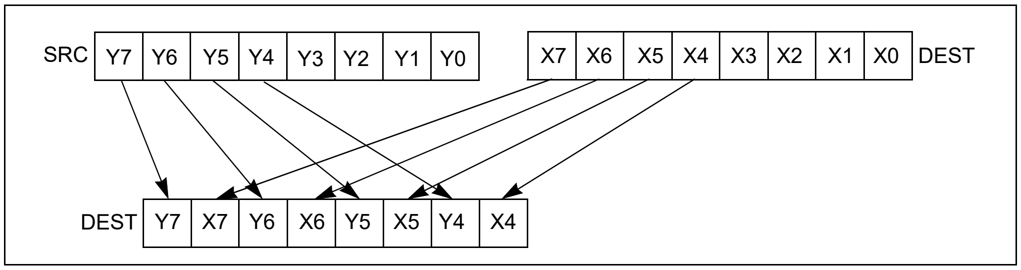


Figure 4-20. PUNPCKHBW Instruction Operation Using 64-bit Operands

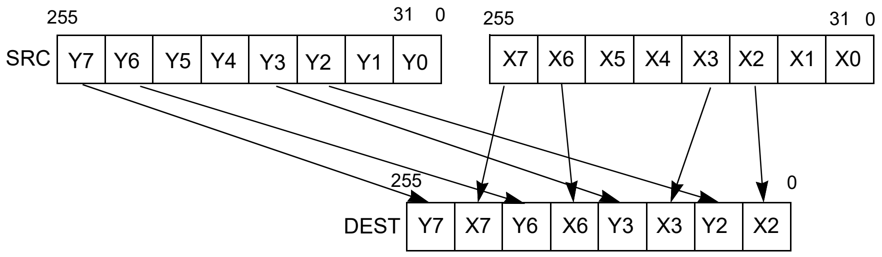


Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKHBW Instruction With 64-bit Operands:

```
    DEST[7:0] := DEST[39:32];
    DEST[15:8] := SRC[39:32];
    DEST[23:16] := DEST[47:40];
    DEST[31:24] := SRC[47:40];
    DEST[39:32] := DEST[55:48];
    DEST[47:40] := SRC[55:48];
    DEST[55:48] := DEST[63:56];
    DEST[63:56] := SRC[63:56];
```

PUNPCKHW Instruction With 64-bit Operands:

```
    DEST[15:0] := DEST[47:32];
    DEST[31:16] := SRC[47:32];
    DEST[47:32] := DEST[63:48];
    DEST[63:48] := SRC[63:48];
```

PUNPCKHDQ Instruction With 64-bit Operands:

```
    DEST[31:0] := DEST[63:32];
    DEST[63:32] := SRC[63:32];

INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)
DEST[7:0] := SRC1[71:64]
DEST[15:8] := SRC2[71:64]
DEST[23:16] := SRC1[79:72]
DEST[31:24] := SRC2[79:72]
DEST[39:32] := SRC1[87:80]
DEST[47:40] := SRC2[87:80]
DEST[55:48] := SRC1[95:88]
DEST[63:56] := SRC2[95:88]
DEST[71:64] := SRC1[103:96]
DEST[79:72] := SRC2[103:96]
DEST[87:80] := SRC1[111:104]
DEST[95:88] := SRC2[111:104]
DEST[103:96] := SRC1[119:112]
DEST[111:104] := SRC2[119:112]
DEST[119:112] := SRC1[127:120]
DEST[127:120] := SRC2[127:120]
DEST[135:128] := SRC1[199:192]
DEST[143:136] := SRC2[199:192]
DEST[151:144] := SRC1[207:200]
DEST[159:152] := SRC2[207:200]
```

```
DEST[167:160] := SRC1[215:208]
DEST[175:168] := SRC2[215:208]
DEST[183:176] := SRC1[223:216]
DEST[191:184] := SRC2[223:216]
DEST[199:192] := SRC1[231:224]
DEST[207:200] := SRC2[231:224]
DEST[215:208] := SRC1[239:232]
DEST[223:216] := SRC2[239:232]
DEST[231:224] := SRC1[247:240]
DEST[239:232] := SRC2[247:240]
DEST[247:240] := SRC1[255:248]
DEST[255:248] := SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] := SRC1[71:64]
DEST[15:8] := SRC2[71:64]
DEST[23:16] := SRC1[79:72]
DEST[31:24] := SRC2[79:72]
DEST[39:32] := SRC1[87:80]
DEST[47:40] := SRC2[87:80]
DEST[55:48] := SRC1[95:88]
DEST[63:56] := SRC2[95:88]
DEST[71:64] := SRC1[103:96]
DEST[79:72] := SRC2[103:96]
DEST[87:80] := SRC1[111:104]
DEST[95:88] := SRC2[111:104]
DEST[103:96] := SRC1[119:112]
DEST[111:104] := SRC2[119:112]
DEST[119:112] := SRC1[127:120]
DEST[127:120] := SRC2[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] := SRC1[79:64]
DEST[31:16] := SRC2[79:64]
DEST[47:32] := SRC1[95:80]
DEST[63:48] := SRC2[95:80]
DEST[79:64] := SRC1[111:96]
DEST[95:80] := SRC2[111:96]
DEST[111:96] := SRC1[127:112]
DEST[127:112] := SRC2[127:112]
DEST[143:128] := SRC1[207:192]
DEST[159:144] := SRC2[207:192]
DEST[175:160] := SRC1[223:208]
DEST[191:176] := SRC2[223:208]
DEST[207:192] := SRC1[239:224]
DEST[223:208] := SRC2[239:224]
DEST[239:224] := SRC1[255:240]
DEST[255:240] := SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
```

```
DEST[15:0] := SRC1[79:64]
DEST[31:16] := SRC2[79:64]
DEST[47:32] := SRC1[95:80]
DEST[63:48] := SRC2[95:80]
DEST[79:64] := SRC1[111:96]
DEST[95:80] := SRC2[111:96]
DEST[111:96] := SRC1[127:112]
DEST[127:112] := SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[159:128] := SRC1[223:192]
DEST[191:160] := SRC2[223:192]
DEST[223:192] := SRC1[255:224]
DEST[255:224] := SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[191:128] := SRC1[255:192]
DEST[255:192] := SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
```

PUNPCKHBW (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHBW (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHBW (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHBW (EVEX Encoded Versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

PUNPCKHWD (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHWD (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHWD (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHWD (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

PUNPCKHDQ (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHDQ (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKHDQ (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHDQ (EVEX.512 Encoded Version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

PUNPCKHQDQ (128-bit Legacy SSE Version)

```
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKHQDQ (VEX.128 Encoded Version)

```
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKHQDQ (VEX.256 Encoded Version)

```
DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKHQDQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*               ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m256i _mm256_mask_unpackhi_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8( __mmask32 k, __m256i a, __m256i b);
```

VPUNPCKHBW __m128i _mm_mask_unpackhi_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKHBW __m128i _mm_maskz_unpackhi_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKHWD __m512i _mm512_unpackhi_epi16(__m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_mask_unpackhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_maskz_unpackhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m256i _mm256_mask_unpackhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m256i _mm256_maskz_unpackhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m128i _mm_mask_unpackhi_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHWD __m128i _mm_maskz_unpackhi_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i _mm512_unpackhi_epi32(__m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_mask_unpackhi_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_unpackhi_epi64(__m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
PUNPCKHBW __m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
(V)PUNPCKHBW __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
VPUNPCKHBW __m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
PUNPCKHWD __m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
(V)PUNPCKHWD __m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
VPUNPCKHWD __m256i _mm256_unpackhi_epi16(__m256i m1,__m256i m2)
PUNPCKHDQ __m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
(V)PUNPCKHDQ __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
VPUNPCKHDQ __m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)
(V)PUNPCKHQDQ __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)
VPUNPCKHQDQ __m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPUNPCKHQDQ/QDQ, see Table 2-52, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 60 /r[1] PUNPCKLBW mm, mm/m32	A	V/V	MMX	Interleave low-order bytes from mm and mm/m32 into mm.
66 0F 60 /r PUNPCKLBW xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.
NP 0F 61 /r[1] PUNPCKLWD mm, mm/m32	A	V/V	MMX	Interleave low-order words from mm and mm/m32 into mm.
66 0F 61 /r PUNPCKLWD xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order words from xmm1 and xmm2/m128 into xmm1.
NP 0F 62 /r[1] PUNPCKLDQ mm, mm/m32	A	V/V	MMX	Interleave low-order doublewords from mm and mm/m32 into mm.
66 0F 62 /r PUNPCKLDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.
66 0F 6C /r PUNPCKLQDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register.
VEX.128.66.0F.WIG 60/r VPUNPCKLBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 61/r VPUNPCKLWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order words from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 62/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1.
VEX.128.66.0F.WIG 6C/r VPUNPCKLQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register.
VEX.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 62 /r VPUNPCKLDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order doublewords from ymm2 and ymm3/m256 into ymm1 register.
VEX.256.66.0F.WIG 6C /r VPUNPCKLQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order quadword from ymm2 and ymm3/m256 into ymm1 register.
EVEX.128.66.0F.WIG 60 /r VPUNPCKLBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.WIG 61 /r VPUNPCKLWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W0 62 /r VPUNPCKLDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W1 6C /r VPUNPCKLQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F OR AVX10.1	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F OR AVX10.1	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

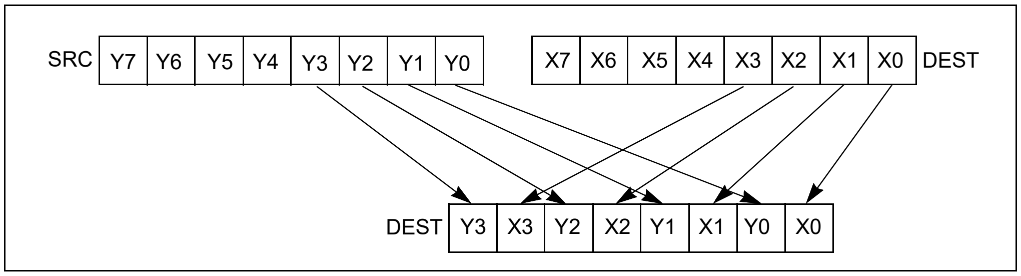


Figure 4-22. PUNPCKLBW Instruction Operation Using 64-bit Operands

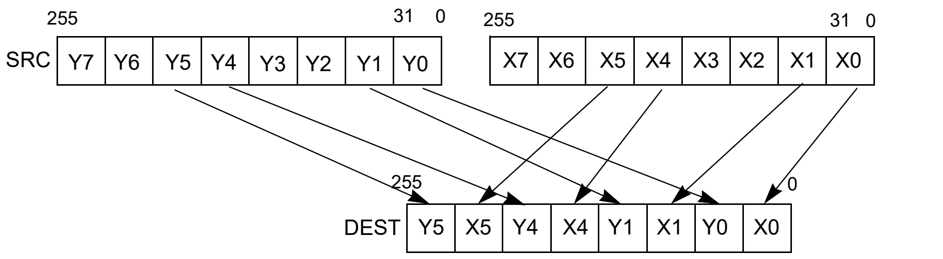


Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNP-CKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKLBW Instruction With 64-bit Operands:

```
    DEST[63:56] := SRC[31:24];
    DEST[55:48] := DEST[31:24];
    DEST[47:40] := SRC[23:16];
    DEST[39:32] := DEST[23:16];
    DEST[31:24] := SRC[15:8];
    DEST[23:16] := DEST[15:8];
    DEST[15:8] := SRC[7:0];
    DEST[7:0] := DEST[7:0];
```

PUNPCKLWD Instruction With 64-bit Operands:

```
    DEST[63:48] := SRC[31:16];
    DEST[47:32] := DEST[31:16];
    DEST[31:16] := SRC[15:0];
    DEST[15:0] := DEST[15:0];
```

PUNPCKLDQ Instruction With 64-bit Operands:

```
    DEST[63:32] := SRC[31:0];
    DEST[31:0] := DEST[31:0];
INTERLEAVE_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_BYTES_256b (SRC1, SRC2)
DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[23:16] := SRC1[15:8]
DEST[31:24] := SRC2[15:8]
DEST[39:32] := SRC1[23:16]
DEST[47:40] := SRC2[23:16]
DEST[55:48] := SRC1[31:24]
DEST[63:56] := SRC2[31:24]
DEST[71:64] := SRC1[39:32]
DEST[79:72] := SRC2[39:32]
DEST[87:80] := SRC1[47:40]
DEST[95:88] := SRC2[47:40]
DEST[103:96] := SRC1[55:48]
DEST[111:104] := SRC2[55:48]
DEST[119:112] := SRC1[63:56]
DEST[127:120] := SRC2[63:56]
DEST[135:128] := SRC1[135:128]
DEST[143:136] := SRC2[135:128]
DEST[151:144] := SRC1[143:136]
DEST[159:152] := SRC2[143:136]
DEST[167:160] := SRC1[151:144]
DEST[175:168] := SRC2[151:144]
```

```
DEST[183:176] := SRC1[159:152]
DEST[191:184] := SRC2[159:152]
DEST[199:192] := SRC1[167:160]
DEST[207:200] := SRC2[167:160]
DEST[215:208] := SRC1[175:168]
DEST[223:216] := SRC2[175:168]
DEST[231:224] := SRC1[183:176]
DEST[239:232] := SRC2[183:176]
DEST[247:240] := SRC1[191:184]
DEST[255:248] := SRC2[191:184]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[23:16] := SRC1[15:8]
DEST[31:24] := SRC2[15:8]
DEST[39:32] := SRC1[23:16]
DEST[47:40] := SRC2[23:16]
DEST[55:48] := SRC1[31:24]
DEST[63:56] := SRC2[31:24]
DEST[71:64] := SRC1[39:32]
DEST[79:72] := SRC2[39:32]
DEST[87:80] := SRC1[47:40]
DEST[95:88] := SRC2[47:40]
DEST[103:96] := SRC1[55:48]
DEST[111:104] := SRC2[55:48]
DEST[119:112] := SRC1[63:56]
DEST[127:120] := SRC2[63:56]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
DEST[47:32] := SRC1[31:16]
DEST[63:48] := SRC2[31:16]
DEST[79:64] := SRC1[47:32]
DEST[95:80] := SRC2[47:32]
DEST[111:96] := SRC1[63:48]
DEST[127:112] := SRC2[63:48]
DEST[143:128] := SRC1[143:128]
DEST[159:144] := SRC2[143:128]
DEST[175:160] := SRC1[159:144]
DEST[191:176] := SRC2[159:144]
DEST[207:192] := SRC1[175:160]
DEST[223:208] := SRC2[175:160]
DEST[239:224] := SRC1[191:176]
DEST[255:240] := SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
```

```
DEST[47:32] := SRC1[31:16]
DEST[63:48] := SRC2[31:16]
DEST[79:64] := SRC1[47:32]
DEST[95:80] := SRC2[47:32]
DEST[111:96] := SRC1[63:48]
DEST[127:112] := SRC2[63:48]

INTERLEAVE_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[159:128] := SRC1[159:128]
DEST[191:160] := SRC2[159:128]
DEST[223:192] := SRC1[191:160]
DEST[255:224] := SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
INTERLEAVE_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[191:128] := SRC1[191:128]
DEST[255:192] := SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
```

PUNPCKLBW

```
DEST[127:0] := INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLBW (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKLBW (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLBW (EVEX.512 Encoded Instruction)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
DEST[511:0] := INTERLEAVE_BYTES_512b(SRC1, SRC2)
```

PUNPCKLWD

```
DEST[127:0] := INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLWD (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0
```

VPUNPCKLWD (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLWD (EVEX.512 Encoded Instruction)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
DEST[511:0] := INTERLEAVE_WORDS_512b(SRC1, SRC2)
```

PUNPCKLDQ

```
DEST[127:0] := INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLDQ (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKLDQ (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLDQ (EVEX Encoded Instructions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST511:0] := INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] := 0
```

PUNPCKLQDQ

```
DEST[127:0] := INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLQDQ (VEX.128 Encoded Instruction)

```
DEST[127:0] := INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0
```

VPUNPCKLQDQ (VEX.256 Encoded Instruction)

```
DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
```

VPUNPCKLQDQ (EVEX Encoded Instructions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m256i _mm256_maskz_unpacklo_epi8( __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i _mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i _mm_maskz_unpacklo_epi8( __mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i _mm512_unpacklo_epi16(__m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_maskz_unpacklo_epi16( __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i _mm256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i _mm256_maskz_unpacklo_epi16( __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i _mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i _mm_maskz_unpacklo_epi16( __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i _mm512_unpacklo_epi32(__m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_maskz_unpacklo_epi32( __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i _mm256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i _mm256_maskz_unpacklo_epi32( __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i _mm_mask_unpacklo_epi32(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i _mm_maskz_unpacklo_epi32( __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i _mm512_unpacklo_epi64(__m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_maskz_unpacklo_epi64( __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i _mm256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i _mm256_maskz_unpacklo_epi64( __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i _mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i _mm_maskz_unpacklo_epi64( __mmask8 k, __m128i a, __m128i b);
PUNPCKLBW __m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
(V)PUNPCKLBW __m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW __m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
PUNPCKLWD __m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
(V)PUNPCKLWD __m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLWD __m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)
PUNPCKLDQ __m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
(V)PUNPCKLDQ __m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
VPUNPCKLDQ __m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)
(V)PUNPCKLQDQ __m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)
VPUNPCKLQDQ __m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPUNPCKLDQ/QDQ, see Table 2-52, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

PXOR—Logical Exclusive OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EF /r[1] PXOR mm, mm/m64	A	V/V	MMX	Bitwise XOR of mm/m64 and mm.
66 0F EF /r PXOR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.
VEX.256.66.0F.WIG EF /r VPXOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise XOR of ymm3/m256 and ymm2.
EVEX.128.66.0F.W0 EF /r VPXORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W0 EF /r VPXORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EF /r VPXORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W1 EF /r VPXORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.5, "Intel® AVX and Intel® SSE Instruction Exception Classification," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, and Section 24.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PXOR (64-bit Operand)

```
DEST := DEST XOR SRC
```

PXOR (128-bit Legacy SSE Version)

```
DEST := DEST XOR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPXOR (VEX.128 Encoded Version)

```
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:128] := 0
```

VPXOR (VEX.256 Encoded Version)

```
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:256] := 0
```

VPXORD (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[31:0]
            ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]
         FI;
   ELSE
      IF *merging-masking*                ; merging-masking
         THEN *DEST[31:0] remains unchanged*
         ELSE                             ; zeroing-masking
            DEST[31:0] := 0
      FI;
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VPXORQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[63:0]
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32( __mmask16 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi32(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi32( __mmask8 m, __m256i a, __m256i b)
VPXORD __m128i _mm_xor_epi32(__m128i a, __m128i b)
VPXORD __m128i _mm_mask_xor_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b)
VPXORD __m128i _mm_maskz_xor_epi32( __mmask16 m, __m128i a, __m128i b)
VPXORQ __m512i _mm512_xor_epi64( __m512i a, __m512i b);
VPXORQ __m512i _mm512_mask_xor_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b);
VPXORQ __m512i _mm512_maskz_xor_epi64(__mmask8 m, __m512i a, __m512i b);
VPXORQ __m256i _mm256_xor_epi64( __m256i a, __m256i b);
VPXORQ __m256i _mm256_mask_xor_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b);
VPXORQ __m256i _mm256_maskz_xor_epi64(__mmask8 m, __m256i a, __m256i b);
VPXORQ __m128i _mm_xor_epi64( __m128i a, __m128i b);
VPXORQ __m128i _mm_mask_xor_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPXORQ __m128i _mm_maskz_xor_epi64(__mmask8 m, __m128i a, __m128i b);
PXOR:__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
(V)PXOR:__m128i _mm_xor_si128 ( __m128i a, __m128i b)
VPXOR:__m256i _mm256_xor_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

RCL/RCR/ROL/ROR—Rotate

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /2	RCL r/m8[2], 1	M1	Valid	Valid	Rotate 9 bits (CF, r/m8) left once.
D2 /2	RCL r/m8[2], CL	MC	Valid	Valid	Rotate 9 bits (CF, r/m8) left CL times.
C0 /2 ib	RCL r/m8[2], imm8	MI	Valid	Valid	Rotate 9 bits (CF, r/m8) left imm8 times.
D1 /2	RCL r/m16, 1	M1	Valid	Valid	Rotate 17 bits (CF, r/m16) left once.
D3 /2	RCL r/m16, CL	MC	Valid	Valid	Rotate 17 bits (CF, r/m16) left CL times.
C1 /2 ib	RCL r/m16, imm8	MI	Valid	Valid	Rotate 17 bits (CF, r/m16) left imm8 times.
D1 /2	RCL r/m32, 1	M1	Valid	Valid	Rotate 33 bits (CF, r/m32) left once.
REX.W + D1 /2	RCL r/m64, 1	M1	Valid	N.E.	Rotate 65 bits (CF, r/m64) left once. Uses a 6 bit count.
D3 /2	RCL r/m32, CL	MC	Valid	Valid	Rotate 33 bits (CF, r/m32) left CL times.
REX.W + D3 /2	RCL r/m64, CL	MC	Valid	N.E.	Rotate 65 bits (CF, r/m64) left CL times. Uses a 6 bit count.
C1 /2 ib	RCL r/m32, imm8	MI	Valid	Valid	Rotate 33 bits (CF, r/m32) left imm8 times.
REX.W + C1 /2 ib	RCL r/m64, imm8	MI	Valid	N.E.	Rotate 65 bits (CF, r/m64) left imm8 times. Uses a 6 bit count.
D0 /3	RCR r/m8[2], 1	M1	Valid	Valid	Rotate 9 bits (CF, r/m8) right once.
D2 /3	RCR r/m8[2], CL	MC	Valid	Valid	Rotate 9 bits (CF, r/m8) right CL times.
C0 /3 ib	RCR r/m8[2], imm8	MI	Valid	Valid	Rotate 9 bits (CF, r/m8) right imm8 times.
D1 /3	RCR r/m16, 1	M1	Valid	Valid	Rotate 17 bits (CF, r/m16) right once.
D3 /3	RCR r/m16, CL	MC	Valid	Valid	Rotate 17 bits (CF, r/m16) right CL times.
C1 /3 ib	RCR r/m16, imm8	MI	Valid	Valid	Rotate 17 bits (CF, r/m16) right imm8 times.
D1 /3	RCR r/m32, 1	M1	Valid	Valid	Rotate 33 bits (CF, r/m32) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR r/m64, 1	M1	Valid	N.E.	Rotate 65 bits (CF, r/m64) right once. Uses a 6 bit count.
D3 /3	RCR r/m32, CL	MC	Valid	Valid	Rotate 33 bits (CF, r/m32) right CL times.
REX.W + D3 /3	RCR r/m64, CL	MC	Valid	N.E.	Rotate 65 bits (CF, r/m64) right CL times. Uses a 6 bit count.
C1 /3 ib	RCR r/m32, imm8	MI	Valid	Valid	Rotate 33 bits (CF, r/m32) right imm8 times.
REX.W + C1 /3 ib	RCR r/m64, imm8	MI	Valid	N.E.	Rotate 65 bits (CF, r/m64) right imm8 times. Uses a 6 bit count.
D0 /0	ROL r/m8,[2] 1	M1	Valid	Valid	Rotate 8 bits r/m8 left once.
D2 /0	ROL r/m8[2], CL	MC	Valid	Valid	Rotate 8 bits r/m8 left CL times.
C0 /0 ib	ROL r/m8[2], imm8	MI	Valid	Valid	Rotate 8 bits r/m8 left imm8 times.
D1 /0	ROL r/m16, 1	M1	Valid	Valid	Rotate 16 bits r/m16 left once.
D3 /0	ROL r/m16, CL	MC	Valid	Valid	Rotate 16 bits r/m16 left CL times.
C1 /0 ib	ROL r/m16, imm8	MI	Valid	Valid	Rotate 16 bits r/m16 left imm8 times.
D1 /0	ROL r/m32, 1	M1	Valid	Valid	Rotate 32 bits r/m32 left once.
REX.W + D1 /0	ROL r/m64, 1	M1	Valid	N.E.	Rotate 64 bits r/m64 left once. Uses a 6 bit count.
D3 /0	ROL r/m32, CL	MC	Valid	Valid	Rotate 32 bits r/m32 left CL times.

Opcode[1]	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
REX.W + D3 /0	ROL r/m64, CL	MC	Valid	N.E.	Rotate 64 bits r/m64 left CL times. Uses a 6 bit count.
C1 /0 ib	ROL r/m32, imm8	MI	Valid	Valid	Rotate 32 bits r/m32 left imm8 times.
REX.W + C1 /0 ib	ROL r/m64, imm8	MI	Valid	N.E.	Rotate 64 bits r/m64 left imm8 times. Uses a 6 bit count.
D0 /1	ROR r/m8[2], 1	M1	Valid	Valid	Rotate 8 bits r/m8 right once.
D2 /1	ROR r/m8[2], CL	MC	Valid	Valid	Rotate 8 bits r/m8 right CL times.
C0 /1 ib	ROR r/m8[2], imm8	MI	Valid	Valid	Rotate 8 bits r/m16 right imm8 times.
D1 /1	ROR r/m16, 1	M1	Valid	Valid	Rotate 16 bits r/m16 right once.
D3 /1	ROR r/m16, CL	MC	Valid	Valid	Rotate 16 bits r/m16 right CL times.
C1 /1 ib	ROR r/m16, imm8	MI	Valid	Valid	Rotate 16 bits r/m16 right imm8 times.
D1 /1	ROR r/m32, 1	M1	Valid	Valid	Rotate 32 bits r/m32 right once.
REX.W + D1 /1	ROR r/m64, 1	M1	Valid	N.E.	Rotate 64 bits r/m64 right once. Uses a 6 bit count.
D3 /1	ROR r/m32, CL	MC	Valid	Valid	Rotate 32 bits r/m32 right CL times.
REX.W + D3 /1	ROR r/m64, CL	MC	Valid	N.E.	Rotate 64 bits r/m64 right CL times. Uses a 6 bit count.
C1 /1 ib	ROR r/m32, imm8	MI	Valid	Valid	Rotate 32 bits r/m32 right imm8 times.
REX.W + C1 /1 ib	ROR r/m64, imm8	MI	Valid	N.E.	Rotate 64 bits r/m64 right imm8 times. Uses a 6 bit count.

NOTES:

1. See the IA-32 Architecture Compatibility section below.
2. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	N/A	N/A
MC	ModRM:r/m (w)	CL	N/A	N/A
MI	ModRM:r/m (w)	imm8	N/A	N/A

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
(* RCL and RCR Instructions *)
SIZE := OperandSize;
CASE (determine count) OF
    SIZE := 8:      tempCOUNT := (COUNT AND 1FH) MOD 9;
    SIZE := 16:     tempCOUNT := (COUNT AND 1FH) MOD 17;
    SIZE := 32:     tempCOUNT := COUNT AND 1FH;
    SIZE := 64:     tempCOUNT := COUNT AND 3FH;
ESAC;
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* RCL Instruction Operation *)
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := MSB(DEST);
        DEST := (DEST ∗ 2) + CF;
        CF := tempCF;
        tempCOUNT := tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
```

```
(* RCR Instruction Operation *)
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := LSB(SRC);
        DEST := (DEST / 2) + (CF * $2^{SIZE}$);
        CF := tempCF;
        tempCOUNT := tempCOUNT – 1;
    OD;

(* ROL Instruction Operation *)
tempCOUNT := (COUNT & COUNTMASK) MOD SIZE

WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := MSB(DEST);
        DEST := (DEST ∗ 2) + tempCF;
        tempCOUNT := tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF := LSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* ROR Instruction Operation *)
tempCOUNT := (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF := LSB(SRC);
        DEST := (DEST / 2) + (tempCF ∗ $2^{SIZE}$);
        tempCOUNT := tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF := MSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF := MSB(DEST) XOR MSB − 1(DEST);
    ELSE OF is undefined;
FI;
```

Flags Affected

For RCL and RCR instructions, a zero-bit rotate does nothing, i.e., affects no flags. For ROL and ROR instructions, if the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined.

For all instructions, the CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

REP—Repeat String Operation (Prefix)

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description[1]
F3 6C	REP INS m8, DX	ZO	Valid	Valid	Input (count) bytes from port DX to (dest).
F3 6D	REP INS m16, DX	ZO	Valid	Valid	Input (count) words from port DX to (dest).
F3 6D	REP INS m32, DX	ZO	Valid	Valid	Input (count) doublewords from port DX to (dest).
F3 AC	REP LODS AL	ZO	Valid	Valid	Load (count) bytes from (src) to AL.
F3 AD	REP LODS AX	ZO	Valid	Valid	Load (count) words from (src) to AX.
F3 AD	REP LODS EAX	ZO	Valid	Valid	Load (count) doublewords from (src) to EAX.
F3 REX.W AD	REP LODS RAX	ZO	Valid	N.E.	Load (count) quadwords from (src) to RAX.
F3 A4	REP MOVS m8, m8	ZO	Valid	Valid	Move (count) bytes from (src) to (dest).
F3 A5	REP MOVS m16, m16	ZO	Valid	Valid	Move (count) words from (src) to (dest).
F3 A5	REP MOVS m32, m32	ZO	Valid	Valid	Move (count) doublewords from (src) to (dest).
F3 REX.W A5	REP MOVS m64, m64	ZO	Valid	N.E.	Move (count) quadwords from (src) to (dest).
F3 6E	REP OUTS DX, m8	ZO	Valid	Valid	Output (count) bytes from (src) to port DX.
F3 6F	REP OUTS DX, m16	ZO	Valid	Valid	Output (count) words from (src) to port DX.
F3 6F	REP OUTS DX, m32	ZO	Valid	Valid	Output (count) doublewords from (src) to port DX.
F3 AA	REP STOS m8	ZO	Valid	Valid	Fill (count) bytes at (dest) with AL.
F3 AB	REP STOS m16	ZO	Valid	Valid	Fill (count) words at (dest) with AX.
F3 AB	REP STOS m32	ZO	Valid	Valid	Fill (count) doublewords at (dest) with EAX.
F3 REX.W AB	REP STOS m64	ZO	Valid	N.E.	Fill (count) quadwords at (dest) with RAX.

NOTES:

1. See Description section for details on (count), (src), and (dest).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Repeats a string instruction the number of times specified in the count register. The count register is CX, ECX, or RCX, depending on the instruction’s address size. The REP (repeat) mnemonic is a prefix that can be added to the INS, OUTS, MOVS, LODS, and STOS instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. The REP prefixes causes the associated instruction to be repeated until the count in register is decremented to 0.

Each of the string instructions uses a source address, a destination address, or both. The source address is DS:SI, DS:ESI, or DS:RSI, depending on the instruction’s address size; the DS segment may be overridden by an instruction prefix. The destination address is ES:DI, ES:EDI, or ES:RDI, depending on the instruction's address size; the ES segment may not be overridden. (Note that, in 64-bit mode, the base addresses of the CS, DS, ES, and SS segments are treated as zero.)

Similarly, the size of the count register is the instruction’s address size. Thus, the default count register in 64-bit mode is RCX; REX.W has no effect on the address size and the count register. If 67H is used to override the default address size, the size of the count register is also overridden.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

REP INS may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g., #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

Operation

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
    Implicit Source/Dest operand for memory use of SI/DI;
  ELSE IF AddressSize = 64
    THEN Use RCX for CountReg;
    Implicit Source/Dest operand for memory use of RSI/RDI;
  ELSE
    Use ECX for CountReg;
    Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
   DO
      Service pending interrupts (if any);
      Execute associated string instruction;
      CountReg := (CountReg – 1);
      IF CountReg = 0
          THEN exit WHILE loop; FI;
      IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
          THEN exit WHILE loop; FI;
   OD;
```

Flags Affected

None.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

REPE/REPZ—Repeat String Operation While Zero (Prefix)

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description[1]
F3 A6	REPE CMPS m8, m8	ZO	Valid	Valid	Find nonmatching bytes in (src1) and (src2).
F3 A7	REPE CMPS m16, m16	ZO	Valid	Valid	Find nonmatching words in (src1) and (src2).
F3 A7	REPE CMPS m32, m32	ZO	Valid	Valid	Find nonmatching doublewords in (src1) and (src2).
F3 REX.W A7	REPE CMPS m64, m64	ZO	Valid	N.E.	Find nonmatching quadwords in (src1) and (src2).
F3 AE	REPE SCAS m8	ZO	Valid	Valid	Find non-AL byte starting at (src2).
F3 AF	REPE SCAS m16	ZO	Valid	Valid	Find non-AX word starting at (src2).
F3 AF	REPE SCAS m32	ZO	Valid	Valid	Find non-EAX doubleword starting at (src2).
F3 REX.W AF	REPE SCAS m64	ZO	Valid	N.E.	Find non-RAX quadword starting at (src2).

NOTES:

1. See Description section for details on (src1) and (src2).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Repeats a string instruction the number of times specified in the count register or until the ZF flag is clear. The count register is CX, ECX, or RCX, depending on the instruction’s address size. The REPE (repeat while equal) and REPZ (repeat while zero) mnemonics are prefixes that can be added to the CMPS and SCAS instructions. (The REPZ prefix is a synonymous form of the REPE prefix.)

The REPE/REPZ prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. These repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0.

The REPE/REPZ prefixes check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not set. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

The ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

Each of the string instructions uses one or two source addresses. The first source address is DS:SI, DS:ESI, or DS:RSI, depending on the instruction’s address size; the DS segment may be overridden by an instruction prefix. The second source address is ES:DI, ES:EDI, or ES:RDI, depending on the instruction's address size; the ES segment may not be overridden. (Note that, in 64-bit mode, the base addresses of the CS, DS, ES, and SS segments are treated as zero.)

Similarly, the size of the count register is the instruction’s address size. Thus, the default count register in 64-bit mode is RCX; REX.W has no effect on the address size and the count register. If 67H is used to override the default address size, the size of the count register is also overridden.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPZ, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Operation

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
        Implicit Source/Dest operand for memory use of SI/DI;
    ELSE IF AddressSize = 64
        THEN Use RCX for CountReg;
        Implicit Source/Dest operand for memory use of RSI/RDI;
    ELSE
        Use ECX for CountReg;
        Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
      DO
            Service pending interrupts (if any);
            Execute associated string instruction;
            CountReg := (CountReg – 1);
            IF ZF = 0
                  THEN exit WHILE loop;
            FI;
      OD;
```

Flags Affected

None by the prefixes; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

REPNE/REPNZ—Repeat String Operation While Not Zero (Prefix)

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description[1]
F2 A6	REPNE CMPS m8, m8	ZO	Valid	Valid	Find matching bytes in (src1) and (src2).
F2 A7	REPNE CMPS m16, m16	ZO	Valid	Valid	Find matching words in (src1) and (src2).
F2 A7	REPNE CMPS m32, m32	ZO	Valid	Valid	Find matching doublewords in (src1) and (src2).
F2 REX.W A7	REPNE CMPS m64, m64	ZO	Valid	N.E.	Find matching quadwords in (src1) and (src2).
F2 AE	REPNE SCAS m8	ZO	Valid	Valid	Find AL, starting at (src2).
F2 AF	REPNE SCAS m16	ZO	Valid	Valid	Find AX, starting at (src2).
F2 AF	REPNE SCAS m32	ZO	Valid	Valid	Find EAX, starting at (src2).
F2 REX.W AF	REPNE SCAS m64	ZO	Valid	N.E.	Find RAX, starting at (src2).

NOTES:

1. See Description section for details on (src1) and (src2).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Repeats a string instruction the number of times specified in the count register or until the ZF flag is set. The count register is CX, ECX, or RCX, depending on the instruction's address size. The REPNE (repeat while not equal) and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to the CMPS and SCAS instructions. (The REPNZ prefix is a synonymous form of the REPNE prefix.)

The REPNE/REPNZ prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. These repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0.

The REPNE/REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is set. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

The ZF flag does not require initialization because it is checked only after each execution of CMPS and SCAS, and those instructions update the ZF flag according to the results of the comparisons they make.

Each of the string instructions uses one or two source addresses. The first source address is DS:SI, DS:ESI, or DS:RSI, depending on the instruction's address size; the DS segment may be overridden by an instruction prefix. The second source address is ES:DI, ES:EDI, or ES:RDI, depending on the instruction's address size; the ES segment may not be overridden. (Note that, in 64-bit mode, the base addresses of the CS, DS, ES, and SS segments are treated as zero.)

Similarly, the size of the count register is the instruction's address size. Thus, the default count register in 64-bit mode is RCX; REX.W has no effect on the address size and the count register. If 67H is used to override the default address size, the size of the count register is also overridden.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPNE or REPNZ, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Operation

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
    Implicit Source/Dest operand for memory use of SI/DI;
  ELSE IF AddressSize = 64
    THEN Use RCX for CountReg;
    Implicit Source/Dest operand for memory use of RSI/RDI;
  ELSE
    Use ECX for CountReg;
    Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
   DO
       Service pending interrupts (if any);
       Execute associated string instruction;
       CountReg := (CountReg – 1);
       IF CountReg = 0
           THEN exit WHILE loop; FI;
       IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
       or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
           THEN exit WHILE loop; FI;
   OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.

SAL/SAR/SHL/SHR—Shift

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL r/m8[2], 1	M1	Valid	Valid	Multiply r/m8 by 2, once.
D2 /4	SAL r/m8[2], CL	MC	Valid	Valid	Multiply r/m8 by 2, CL times.
C0 /4 ib	SAL r/m8[2], imm8	MI	Valid	Valid	Multiply r/m8 by 2, imm8 times.
D1 /4	SAL r/m16, 1	M1	Valid	Valid	Multiply r/m16 by 2, once.
D3 /4	SAL r/m16, CL	MC	Valid	Valid	Multiply r/m16 by 2, CL times.
C1 /4 ib	SAL r/m16, imm8	MI	Valid	Valid	Multiply r/m16 by 2, imm8 times.
D1 /4	SAL r/m32, 1	M1	Valid	Valid	Multiply r/m32 by 2, once.
REX.W + D1 /4	SAL r/m64, 1	M1	Valid	N.E.	Multiply r/m64 by 2, once.
D3 /4	SAL r/m32, CL	MC	Valid	Valid	Multiply r/m32 by 2, CL times.
REX.W + D3 /4	SAL r/m64, CL	MC	Valid	N.E.	Multiply r/m64 by 2, CL times.
C1 /4 ib	SAL r/m32, imm8	MI	Valid	Valid	Multiply r/m32 by 2, imm8 times.
REX.W + C1 /4 ib	SAL r/m64, imm8	MI	Valid	N.E.	Multiply r/m64 by 2, imm8 times.
D0 /7	SAR r/m8[2], 1	M1	Valid	Valid	Signed divide[3] r/m8 by 2, once.
D2 /7	SAR r/m8[2], CL	MC	Valid	Valid	Signed divide[3] r/m8 by 2, CL times.
C0 /7 ib	SAR r/m8[2], imm8	MI	Valid	Valid	Signed divide[3] r/m8 by 2, imm8 times.
D1 /7	SAR r/m16,1	M1	Valid	Valid	Signed divide[3] r/m16 by 2, once.
D3 /7	SAR r/m16, CL	MC	Valid	Valid	Signed divide[3] r/m16 by 2, CL times.
C1 /7 ib	SAR r/m16, imm8	MI	Valid	Valid	Signed divide[3] r/m16 by 2, imm8 times.
D1 /7	SAR r/m32, 1	M1	Valid	Valid	Signed divide[3] r/m32 by 2, once.
REX.W + D1 /7	SAR r/m64, 1	M1	Valid	N.E.	Signed divide[3] r/m64 by 2, once.
D3 /7	SAR r/m32, CL	MC	Valid	Valid	Signed divide[3] r/m32 by 2, CL times.
REX.W + D3 /7	SAR r/m64, CL	MC	Valid	N.E.	Signed divide[3] r/m64 by 2, CL times.
C1 /7 ib	SAR r/m32, imm8	MI	Valid	Valid	Signed divide[3] r/m32 by 2, imm8 times.
REX.W + C1 /7 ib	SAR r/m64, imm8	MI	Valid	N.E.	Signed divide[3] r/m64 by 2, imm8 times
D0 /4	SHL r/m8[2], 1	M1	Valid	Valid	Multiply r/m8 by 2, once.
D2 /4	SHL r/m8[2], CL	MC	Valid	Valid	Multiply r/m8 by 2, CL times.
C0 /4 ib	SHL r/m8[2], imm8	MI	Valid	Valid	Multiply r/m8 by 2, imm8 times.
D1 /4	SHL r/m16,1	M1	Valid	Valid	Multiply r/m16 by 2, once.
D3 /4	SHL r/m16, CL	MC	Valid	Valid	Multiply r/m16 by 2, CL times.
C1 /4 ib	SHL r/m16, imm8	MI	Valid	Valid	Multiply r/m16 by 2, imm8 times.
D1 /4	SHL r/m32,1	M1	Valid	Valid	Multiply r/m32 by 2, once.
REX.W + D1 /4	SHL r/m64,1	M1	Valid	N.E.	Multiply r/m64 by 2, once.
D3 /4	SHL r/m32, CL	MC	Valid	Valid	Multiply r/m32 by 2, CL times.
REX.W + D3 /4	SHL r/m64, CL	MC	Valid	N.E.	Multiply r/m64 by 2, CL times.
C1 /4 ib	SHL r/m32, imm8	MI	Valid	Valid	Multiply r/m32 by 2, imm8 times.
REX.W + C1 /4 ib	SHL r/m64, imm8	MI	Valid	N.E.	Multiply r/m64 by 2, imm8 times.
D0 /5	SHR r/m8[2],1	M1	Valid	Valid	Unsigned divide r/m8 by 2, once.
D2 /5	SHR r/m8[2], CL	MC	Valid	Valid	Unsigned divide r/m8 by 2, CL times.
C0 /5 ib	SHR r/m8[2], imm8	MI	Valid	Valid	Unsigned divide r/m8 by 2, imm8 times.
D1 /5	SHR r/m16, 1	M1	Valid	Valid	Unsigned divide r/m16 by 2, once.

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D3 /5	SHR r/m16, CL	MC	Valid	Valid	Unsigned divide r/m16 by 2, CL times
C1 /5 ib	SHR r/m16, imm8	MI	Valid	Valid	Unsigned divide r/m16 by 2, imm8 times.
D1 /5	SHR r/m32, 1	M1	Valid	Valid	Unsigned divide r/m32 by 2, once.
REX.W + D1 /5	SHR r/m64, 1	M1	Valid	N.E.	Unsigned divide r/m64 by 2, once.
D3 /5	SHR r/m32, CL	MC	Valid	Valid	Unsigned divide r/m32 by 2, CL times.
REX.W + D3 /5	SHR r/m64, CL	MC	Valid	N.E.	Unsigned divide r/m64 by 2, CL times.
C1 /5 ib	SHR r/m32, imm8	MI	Valid	Valid	Unsigned divide r/m32 by 2, imm8 times.
REX.W + C1 /5 ib	SHR r/m64, imm8	MI	Valid	N.E.	Unsigned divide r/m64 by 2, imm8 times.

NOTES:

1. See the IA-32 Architecture Compatibility section below.
2. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.
3. Not the same form of division as IDIV; rounding is toward negative infinity.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (r, w)	1	N/A	N/A
MC	ModRM:r/m (r, w)	CL	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits with a 64-bit operand). The count range is limited to 0 to 31 (or 63 with a 64-bit operand). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position’s shifted value with the sign of the unshifted value (see Figure 7-9 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the “quotient” of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the “remainder” is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction’s default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
IF OperandSize = 64
    THEN
        countMASK := 3FH;
    ELSE
        countMASK := 1FH;
FI
tempCOUNT := (COUNT AND countMASK);
tempDEST := DEST;
WHILE (tempCOUNT ≠ 0)
DO
    IF instruction is SAL or SHL
        THEN
            CF := MSB(DEST);
        ELSE (* Instruction is SAR or SHR *)
            CF := LSB(DEST);
    FI;
    IF instruction is SAL or SHL
        THEN
            DEST := DEST ∗ 2;
        ELSE
            IF instruction is SAR
                THEN
                    DEST := DEST / 2; (* Signed divide, rounding toward negative infinity *)
                ELSE (* Instruction is SHR *)
                    DEST := DEST / 2 ; (* Unsigned divide *)
            FI;
    FI;
    tempCOUNT := tempCOUNT – 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF := MSB(DEST) XOR CF;
                ELSE
                    IF instruction is SAR
                        THEN
                            OF := 0;
                        ELSE (* Instruction is SHR *)
                            OF := MSB(tempDEST);
                    FI;
            FI;
    ELSE IF (COUNT AND countMASK) = 0
        THEN
            All flags unchanged;
        ELSE (* COUNT not 1 or 0 *)
            OF := undefined;
    FI;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SBB—Integer Subtraction With Borrow

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
1C ib	SBB AL, imm8	I	Valid	Valid	Subtract with borrow imm8 from AL.
1D iw	SBB AX, imm16	I	Valid	Valid	Subtract with borrow imm16 from AX.
1D id	SBB EAX, imm32	I	Valid	Valid	Subtract with borrow imm32 from EAX.
REX.W + 1D id	SBB RAX, imm32	I	Valid	N.E.	Subtract with borrow sign-extended imm.32 to 64-bits from RAX.
80 /3 ib	SBB r/m8[1], imm8	MI	Valid	Valid	Subtract with borrow imm8 from r/m8.
81 /3 iw	SBB r/m16, imm16	MI	Valid	Valid	Subtract with borrow imm16 from r/m16.
81 /3 id	SBB r/m32, imm32	MI	Valid	Valid	Subtract with borrow imm32 from r/m32.
REX.W + 81 /3 id	SBB r/m64, imm32	MI	Valid	N.E.	Subtract with borrow sign-extended imm32 to 64-bits from r/m64.
83 /3 ib	SBB r/m16, imm8	MI	Valid	Valid	Subtract with borrow sign-extended imm8 from r/m16.
83 /3 ib	SBB r/m32, imm8	MI	Valid	Valid	Subtract with borrow sign-extended imm8 from r/m32.
REX.W + 83 /3 ib	SBB r/m64, imm8	MI	Valid	N.E.	Subtract with borrow sign-extended imm8 from r/m64.
18 /r	SBB r/m8[1], r8[1]	MR	Valid	Valid	Subtract with borrow r8 from r/m8.
19 /r	SBB r/m16, r16	MR	Valid	Valid	Subtract with borrow r16 from r/m16.
19 /r	SBB r/m32, r32	MR	Valid	Valid	Subtract with borrow r32 from r/m32.
REX.W + 19 /r	SBB r/m64, r64	MR	Valid	N.E.	Subtract with borrow r64 from r/m64.
1A /r	SBB r8[1], r/m8[1]	RM	Valid	Valid	Subtract with borrow r/m8 from r8.
1B /r	SBB r16, r/m16	RM	Valid	Valid	Subtract with borrow r/m16 from r16.
1B /r	SBB r32, r/m32	RM	Valid	Valid	Subtract with borrow r/m32 from r32.
REX.W + 1B /r	SBB r64, r/m64	RM	Valid	N.E.	Subtract with borrow r/m64 from r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := (DEST – (SRC + CF));
```

Intel C/C++ Compiler Intrinsic Equivalent

```
SBB extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);
SBB extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);
SBB extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *diff_out);
SBB extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64
*diff_out);
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SETcc—Set Byte on Condition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA r/m8[1]	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE r/m8[1]	M	Valid	Valid	Set byte if above or equal (CF=0).
0F 92	SETB r/m8[1]	M	Valid	Valid	Set byte if below (CF=1).
0F 96	SETBE r/m8[1]	M	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC r/m8[1]	M	Valid	Valid	Set byte if carry (CF=1).
0F 94	SETE r/m8[1]	M	Valid	Valid	Set byte if equal (ZF=1).
0F 9F	SETG r/m8[1]	M	Valid	Valid	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE r/m8[1]	M	Valid	Valid	Set byte if greater or equal (SF=OF).
0F 9C	SETL r/m8[1]	M	Valid	Valid	Set byte if less (SF≠ OF).
0F 9E	SETLE r/m8[1]	M	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ OF).
0F 96	SETNA r/m8[1]	M	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE r/m8[1]	M	Valid	Valid	Set byte if not above or equal (CF=1).
0F 93	SETNB r/m8[1]	M	Valid	Valid	Set byte if not below (CF=0).
0F 97	SETNBE r/m8[1]	M	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC r/m8[1]	M	Valid	Valid	Set byte if not carry (CF=0).
0F 95	SETNE r/m8[1]	M	Valid	Valid	Set byte if not equal (ZF=0).
0F 9E	SETNG r/m8[1]	M	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ OF)
0F 9C	SETNGE r/m8[1]	M	Valid	Valid	Set byte if not greater or equal (SF≠ OF).
0F 9D	SETNL r/m8[1]	M	Valid	Valid	Set byte if not less (SF=OF).
0F 9F	SETNLE r/m8[1]	M	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO r/m8[1]	M	Valid	Valid	Set byte if not overflow (OF=0).
0F 9B	SETNP r/m8[1]	M	Valid	Valid	Set byte if not parity (PF=0).
0F 99	SETNS r/m8[1]	M	Valid	Valid	Set byte if not sign (SF=0).
0F 95	SETNZ r/m8[1]	M	Valid	Valid	Set byte if not zero (ZF=0).
0F 90	SETO r/m8[1]	M	Valid	Valid	Set byte if overflow (OF=1)
0F 9A	SETP r/m8[1]	M	Valid	Valid	Set byte if parity (PF=1).
0F 9A	SETPE r/m8[1]	M	Valid	Valid	Set byte if parity even (PF=1).
0F 9B	SETPO r/m8[1]	M	Valid	Valid	Set byte if parity odd (PF=0).
0F 98	SETS r/m8[1]	M	Valid	Valid	Set byte if sign (SF=1).
0F 94	SETZ r/m8[1]	M	Valid	Valid	Set byte if zero (ZF=1).

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms "above" and "below" are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, "EFLAGS Condition Codes," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

The reg field of the ModR/M byte is not used for the SETCC instruction and those opcode bits are ignored by the processor.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction's operation is the same as in legacy mode and compatibility mode.

Operation

```
IF condition
    THEN DEST := 1;
    ELSE DEST := 0;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

SHUFPD—Packed Interleave Shuffle of Pairs of Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle two pairs of double precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in xmm1.
EVEX.128.66.0F.W1 C6 /r ib VSHUFPD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 C6 /r ib VSHUFPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 C6 /r ib VSHUFPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F OR AVX10.1	Shuffle eight pairs of double precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Selects a double precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4) are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

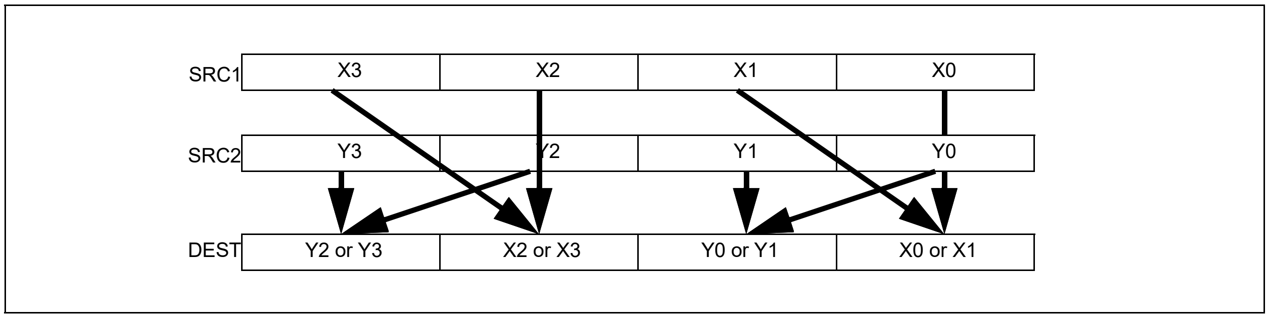


Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of Double Precision Floating-Point Values

Operation

VSHUFPD (EVEX Encoded Versions When SRC2 is a Vector Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] := SRC1[63:0]
    ELSE TMP_DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] := SRC2[63:0]
    ELSE TMP_DEST[127:64] := SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] := SRC1[191:128]
        ELSE TMP_DEST[191:128] := SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] := SRC2[191:128]
        ELSE TMP_DEST[255:192] := SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] := SRC1[319:256]
        ELSE TMP_DEST[319:256] := SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] := SRC2[319:256]
        ELSE TMP_DEST[383:320] := SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] := SRC1[447:384]
        ELSE TMP_DEST[447:384] := SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] := SRC2[447:384]
        ELSE TMP_DEST[511:448] := SRC2[511:448] FI;
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPD (EVEX Encoded Versions When SRC2 is Memory)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] := SRC1[63:0]
    ELSE TMP_DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] := TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] := TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] := SRC1[191:128]
        ELSE TMP_DEST[191:128] := SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] := TMP_SRC2[191:128]
        ELSE TMP_DEST[255:192] := TMP_SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] := SRC1[319:256]
        ELSE TMP_DEST[319:256] := SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] := TMP_SRC2[319:256]
        ELSE TMP_DEST[383:320] := TMP_SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] := SRC1[447:384]
        ELSE TMP_DEST[447:384] := SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] := TMP_SRC2[447:384]
        ELSE TMP_DEST[511:448] := TMP_SRC2[511:448] FI;
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
```

```
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPD (VEX.256 Encoded Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] := SRC1[191:128]
    ELSE DEST[191:128] := SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] := SRC2[191:128]
    ELSE DEST[255:192] := SRC2[255:192] FI;
DEST[MAXVL-1:256] (Unmodified)
```

VSHUFPD (VEX.128 Encoded Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] := 0
```

VSHUFPD (128-bit Legacy SSE Version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] := SRC2[63:0]
    ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPD __m512d _mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);
VSHUFPD __m256d _mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d _mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);
VSHUFPD __m128d _mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d _mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

SHUFPS—Packed Interleave Shuffle of Quadruplets of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	A	V/V	SSE	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select from quadruplet of single precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.128.0F.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.256.0F.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Select from quadruplet of single precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	C	V/V	AVX512F OR AVX10.1	Select from quadruplet of single precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Selects a single precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

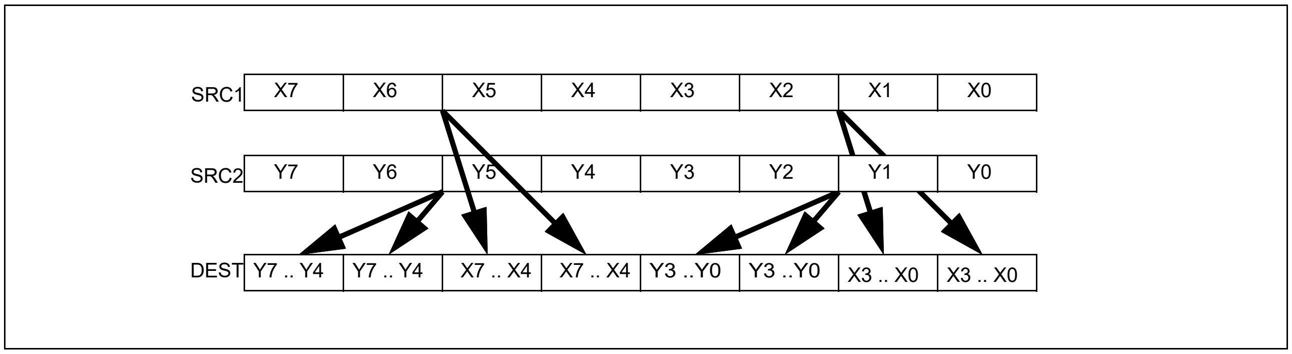


Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[31:0];
    1:    TMP := SRC[63:32];
    2:    TMP := SRC[95:64];
    3:    TMP := SRC[127:96];
ESAC;
RETURN TMP
}
```

VPSHUFPS (EVEX Encoded Versions When SRC2 is a Vector Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP_DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] := Select4(SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] := Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] := Select4(SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] := Select4(SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] := Select4(SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] := Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
```

```
            THEN DEST[i+31:i] := TMP_DEST[i+31:i]
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPSHUFPS (EVEX Encoded Versions When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
TMP_DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] := Select4(TMP_SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] := Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] := Select4(TMP_SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] := Select4(TMP_SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] := Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] := Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFPS (VEX.256 Encoded Version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] := Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] := Select4(SRC2[255:128], imm8[7:6]);
DEST[MAXVL-1:256] := 0
```

VSHUFPS (VEX.128 Encoded Version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0
```

SHUFPS (128-bit Legacy SSE Version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

SQRTPD—Square Root of Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	A	V/V	SSE2	Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed double precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Computes Square Roots of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a SIMD computation of the square roots of the two, four or eight packed double precision floating-point values in the source operand (the second operand) stores the packed double precision floating-point results in the destination operand (the first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC *is memory*)
                    THEN DEST[i+63:i] := SQRT(SRC[63:0])
                    ELSE DEST[i+63:i] := SQRT(SRC[i+63:i])
                FI;
        ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSQRTPD (VEX.256 Encoded Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[191:128] := SQRT(SRC[191:128])
DEST[255:192] := SQRT(SRC[255:192])
DEST[MAXVL-1:256] := 0
.
```

VSQRTPD (VEX.128 Encoded Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] := 0
```

SQRTPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTPD __m512d _mm512_sqrt_round_pd(__m512d a, int r);
VSQRTPD __m512d _mm512_mask_sqrt_round_pd(__m512d s, __mmask8 k, __m512d a, int r);
VSQRTPD __m512d _mm512_maskz_sqrt_round_pd( __mmask8 k, __m512d a, int r);
VSQRTPD __m256d _mm256_sqrt_pd (__m256d a);
VSQRTPD __m256d _mm256_mask_sqrt_pd(__m256d s, __mmask8 k, __m256d a, int r);
VSQRTPD __m256d _mm256_maskz_sqrt_pd( __mmask8 k, __m256d a, int r);
SQRTPD __m128d _mm_sqrt_pd (__m128d a);
VSQRTPD __m128d _mm_mask_sqrt_pd(__m128d s, __mmask8 k, __m128d a, int r);
VSQRTPD __m128d _mm_maskz_sqrt_pd( __mmask8 k, __m128d a, int r);
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions,” additionally:

#UD	If EVEX.vvvv != 1111B.

SQRTPS—Square Root of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 51 /r SQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed single precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.0F.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed single precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes Square Roots of the packed single precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Computes Square Roots of the packed single precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single precision floating-point values in the source operand (second operand) stores the packed single precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC *is memory*)
                    THEN DEST[i+31:i] := SQRT(SRC[31:0])
                    ELSE DEST[i+31:i] := SQRT(SRC[i+31:i])
                FI;
        ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSQRTPS (VEX.256 Encoded Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[159:128] := SQRT(SRC[159:128])
DEST[191:160] := SQRT(SRC[191:160])
DEST[223:192] := SQRT(SRC[223:192])
DEST[255:224] := SQRT(SRC[255:224])
```

VSQRTPS (VEX.128 Encoded Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[MAXVL-1:128] := 0
```

SQRTPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SQRT(SRC[31:0])
DEST[63:32] := SQRT(SRC[63:32])
DEST[95:64] := SQRT(SRC[95:64])
DEST[127:96] := SQRT(SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps (__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions,” additionally:

#UD	If EVEX.vvvv != 1111B.

SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	A	V/V	SSE2	Computes square root of the low double precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.LIG.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.LLIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the square root of the low double precision floating-point value in the second source operand and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSD (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SQRT(SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VSQRTSD (VEX.128 Encoded Version)

```
DEST[63:0] := SQRT(SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

SQRTSD (128-bit Legacy SSE Version)

```
DEST[63:0] := SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

SQRTSS—Compute Square Root of Scalar Single Precision Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the square root of the low single precision floating-point value in the second source operand and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the write-mask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSS (EVEX Encoded Version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SQRT(SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VSQRTSS (VEX.128 Encoded Version)

```
DEST[31:0] := SQRT(SRC2[31:0])
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

SQRTSS (128-bit Legacy SSE Version)

```
DEST[31:0] := SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss( __mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E3 Class Exception Conditions.”

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C ib	SUB AL, imm8	I	Valid	Valid	Subtract imm8 from AL.
2D iw	SUB AX, imm16	I	Valid	Valid	Subtract imm16 from AX.
2D id	SUB EAX, imm32	I	Valid	Valid	Subtract imm32 from EAX.
REX.W + 2D id	SUB RAX, imm32	I	Valid	N.E.	Subtract imm32 sign-extended to 64-bits from RAX.
80 /5 ib	SUB r/m8[1], imm8	MI	Valid	Valid	Subtract imm8 from r/m8.
81 /5 iw	SUB r/m16, imm16	MI	Valid	Valid	Subtract imm16 from r/m16.
81 /5 id	SUB r/m32, imm32	MI	Valid	Valid	Subtract imm32 from r/m32.
REX.W + 81 /5 id	SUB r/m64, imm32	MI	Valid	N.E.	Subtract imm32 sign-extended to 64-bits from r/m64.
83 /5 ib	SUB r/m16, imm8	MI	Valid	Valid	Subtract sign-extended imm8 from r/m16.
83 /5 ib	SUB r/m32, imm8	MI	Valid	Valid	Subtract sign-extended imm8 from r/m32.
REX.W + 83 /5 ib	SUB r/m64, imm8	MI	Valid	N.E.	Subtract sign-extended imm8 from r/m64.
28 /r	SUB r/m8[1], r8[1]	MR	Valid	Valid	Subtract r8 from r/m8.
29 /r	SUB r/m16, r16	MR	Valid	Valid	Subtract r16 from r/m16.
29 /r	SUB r/m32, r32	MR	Valid	Valid	Subtract r32 from r/m32.
REX.W + 29 /r	SUB r/m64, r64	MR	Valid	N.E.	Subtract r64 from r/m64.
2A /r	SUB r8[1], r/m8[1]	RM	Valid	Valid	Subtract r/m8 from r8.
2B /r	SUB r16, r/m16	RM	Valid	Valid	Subtract r/m16 from r16.
2B /r	SUB r32, r/m32	RM	Valid	Valid	Subtract r/m32 from r32.
REX.W + 2B /r	SUB r64, r/m64	RM	Valid	N.E.	Subtract r/m64 from r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST := (DEST – SRC);
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SUBPD—Subtract Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/E n	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed double precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed double precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed double precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 5C /r VSUBPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.66.0F.W1 5C /r VSUBPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.66.0F.W1 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F OR AVX10.1	Subtract packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the two, four or eight packed double precision floating-point values of the second Source operand from the first Source operand, and stores the packed double precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPD (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] := 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSUBPD (EVEX Encoded Versions When SRC2 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1)
                THEN    DEST[i+63:i] := SRC1[i+63:i] - SRC2[63:0];
                ELSE    EST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i];
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] := 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSUBPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64] - SRC2[127:64]
DEST[191:128] := SRC1[191:128] - SRC2[191:128]
DEST[255:192] := SRC1[255:192] - SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VSUBPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64] - SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

SUBPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[127:64] := DEST[127:64] - SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBPD __m512d _mm512_sub_pd (__m512d a, __m512d b);
VSUBPD __m512d _mm512_mask_sub_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_maskz_sub_pd (__mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_sub_round_pd (__m512d a, __m512d b, int);
VSUBPD __m512d _mm512_mask_sub_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m512d _mm512_maskz_sub_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);
VSUBPD __m256d _mm256_mask_sub_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VSUBPD __m256d _mm256_maskz_sub_pd (__mmask8 k, __m256d a, __m256d b);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);
VSUBPD __m128d _mm_mask_sub_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBPD __m128d _mm_maskz_sub_pd (__mmask8 k, __m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

SUBPS—Subtract Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/E n	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5C /r SUBPS xmm1, xmm2/m128	A	V/V	SSE	Subtract packed single precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed single precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed single precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.128.0F.W0 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.256.0F.W0 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Subtract packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F OR AVX10.1	Subtract packed single precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD subtract of the packed single precision floating-point values in the second Source operand from the First Source operand, and stores the packed single precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPS (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VSUBPS (EVEX Encoded Versions When SRC2 Operand is a Memory Source)

```
(KL, VL) = (4, 128), (8, 256),(16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1)
                THEN    DEST[i+31:i] := SRC1[i+31:i] - SRC2[31:0];
                ELSE    DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i];
            FI;

    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VSUBPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[159:128] := SRC1[159:128] - SRC2[159:128]
DEST[191:160] := SRC1[191:160] - SRC2[191:160]
DEST[223:192] := SRC1[223:192] - SRC2[223:192]
DEST[255:224] := SRC1[255:224] - SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VSUBPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

SUBPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[63:32] := SRC1[63:32] - SRC2[63:32]
DEST[95:64] := SRC1[95:64] - SRC2[95:64]
DEST[127:96] := SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBPS __m512 _mm512_sub_ps (__m512 a, __m512 b);
VSUBPS __m512 _mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_sub_round_ps (__m512 a, __m512 b, int);
VSUBPS __m512 _mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m512 _mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);
VSUBPS __m256 _mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VSUBPS __m256 _mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);
SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);
VSUBPS __m128 _mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBPS __m128 _mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

SUBSD—Subtract Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD xmm1, xmm2/m64	A	V/V	SSE2	Subtract the low double precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.LIG.F2.0F.WIG 5C /r VSUBSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.LLIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F OR AVX10.1	Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Subtract the low double precision floating-point value in the second source operand from the first source operand and stores the double precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the write-mask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VSUBSD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

SUBSD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

SUBSS—Subtract Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	A	V/V	SSE	Subtract the low single precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Subtract the low single precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.LLIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F OR AVX10.1	Subtract the low single precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Subtract the low single precision floating-point value from the second source operand and the first source operand and store the double precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

VSUBSS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

SUBSS (128-bit Legacy SSE Version)

```
DEST[31:0] := DEST[31:0] - SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

TEST—Logical Compare

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A8 ib	TEST AL, imm8	I	Valid	Valid	AND imm8 with AL; set SF, ZF, PF according to result.
A9 iw	TEST AX, imm16	I	Valid	Valid	AND imm16 with AX; set SF, ZF, PF according to result.
A9 id	TEST EAX, imm32	I	Valid	Valid	AND imm32 with EAX; set SF, ZF, PF according to result.
REX.W + A9 id	TEST RAX, imm32	I	Valid	N.E.	AND imm32 sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 ib	TEST r/m8[1], imm8	MI	Valid	Valid	AND imm8 with r/m8; set SF, ZF, PF according to result.
F7 /0 iw	TEST r/m16, imm16	MI	Valid	Valid	AND imm16 with r/m16; set SF, ZF, PF according to result.
F7 /0 id	TEST r/m32, imm32	MI	Valid	Valid	AND imm32 with r/m32; set SF, ZF, PF according to result.
REX.W + F7 /0 id	TEST r/m64, imm32	MI	Valid	N.E.	AND imm32 sign-extended to 64-bits with r/m64; set SF, ZF, PF according to result.
84 /r	TEST r/m8[1], r8[1]	MR	Valid	Valid	AND r8 with r/m8; set SF, ZF, PF according to result.
85 /r	TEST r/m16, r16	MR	Valid	Valid	AND r16 with r/m16; set SF, ZF, PF according to result.
85 /r	TEST r/m32, r32	MR	Valid	Valid	AND r32 with r/m32; set SF, ZF, PF according to result.
REX.W + 85 /r	TEST r/m64, r64	MR	Valid	N.E.	AND r64 with r/m64; set SF, ZF, PF according to result.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r)	ModRM:reg (r)	N/A	N/A

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
TEMP := SRC1 AND SRC2;
SF := MSB(TEMP);
IF TEMP = 0
    THEN ZF := 1;
    ELSE ZF := 0;
FI:
PF := BitwiseXNOR(TEMP[0:7]);
CF := 0;
OF := 0;
```

```
(* AF is undefined *)
```

Flags Affected

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

UCOMISD—Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LLIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F OR AVX10.1	Compare low double precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Performs an unordered compare of the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid operation exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISD (All Versions)

```
RESULT := UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int _mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

UCOMISS—Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2E /r UCOMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2E /r VUCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LLIG.0F.W0 2E /r VUCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F OR AVX10.1	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Compares the single precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid operation exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISS (All Versions)

```
RESULT := UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUCOMISS    int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS     int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS     int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS     int _mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS     int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS     int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS     int _mm_ucomineq_ss(__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 15 /r VUNPCKHPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 15 /r VUNPCKHPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 15 /r VUNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves double precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the high double precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPD (EVEX Encoded Versions When SRC2 is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[127:64]
    TMP_DEST[127:64] := SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[255:192]
    TMP_DEST[255:192] := SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[383:320]
    TMP_DEST[383:320] := SRC2[383:320]
    TMP_DEST[447:384] := SRC1[511:448]
    TMP_DEST[511:448] := SRC2[511:448]
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPD (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] := SRC1[127:64]
    TMP_DEST[127:64] := TMP_SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[255:192]
    TMP_DEST[255:192] := TMP_SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[383:320]
    TMP_DEST[383:320] := TMP_SRC2[383:320]
    TMP_DEST[447:384] := SRC1[511:448]
    TMP_DEST[511:448] := TMP_SRC2[511:448]
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[191:128] := SRC1[255:192]
DEST[255:192] := SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VUNPCKHPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

UNPCKHPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKHPD __m512d _mm512_unpackhi_pd( __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-52, “Type E4NF Class Exception Conditions.”

UNPCKHPS—Unpack and Interleave High Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 15 /r UNPCKHPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.256.0F.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves single precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the high single precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

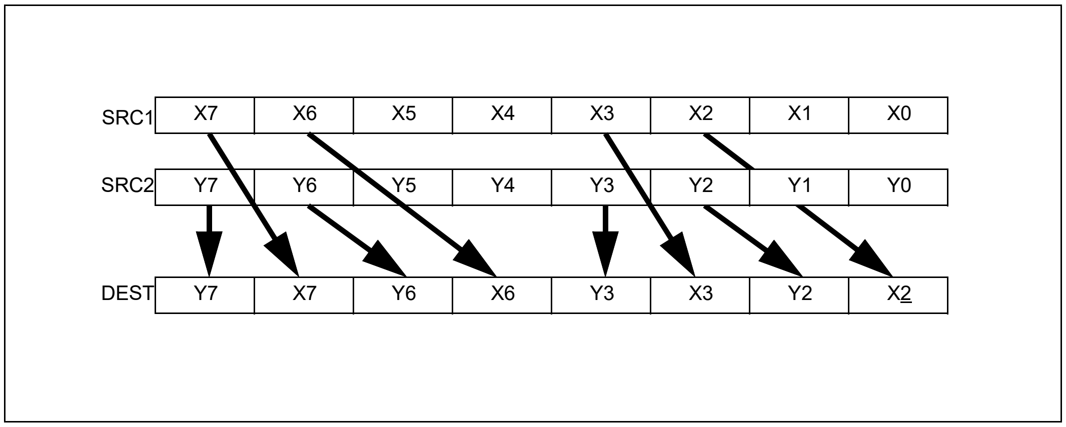


Figure 4-27. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPS (EVEX Encoded Version When SRC2 is a Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] := SRC1[95:64]
    TMP_DEST[63:32] := SRC2[95:64]
    TMP_DEST[95:64] := SRC1[127:96]
    TMP_DEST[127:96] := SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[223:192]
    TMP_DEST[191:160] := SRC2[223:192]
    TMP_DEST[223:192] := SRC1[255:224]
    TMP_DEST[255:224] := SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[351:320]
    TMP_DEST[319:288] := SRC2[351:320]
    TMP_DEST[351:320] := SRC1[383:352]
    TMP_DEST[383:352] := SRC2[383:352]
    TMP_DEST[415:384] := SRC1[479:448]
    TMP_DEST[447:416] := SRC2[479:448]
    TMP_DEST[479:448] := SRC1[511:480]
    TMP_DEST[511:480] := SRC2[511:480]
FI;
```

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPS (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] := SRC1[95:64]
    TMP_DEST[63:32] := TMP_SRC2[95:64]
    TMP_DEST[95:64] := SRC1[127:96]
    TMP_DEST[127:96] := TMP_SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[223:192]
    TMP_DEST[191:160] := TMP_SRC2[223:192]
    TMP_DEST[223:192] := SRC1[255:224]
    TMP_DEST[255:224] := TMP_SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[351:320]
    TMP_DEST[319:288] := TMP_SRC2[351:320]
    TMP_DEST[351:320] := SRC1[383:352]
    TMP_DEST[383:352] := TMP_SRC2[383:352]
    TMP_DEST[415:384] := SRC1[479:448]
    TMP_DEST[447:416] := TMP_SRC2[479:448]
    TMP_DEST[479:448] := SRC1[511:480]
    TMP_DEST[511:480] := TMP_SRC2[511:480]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKHPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[159:128] := SRC1[223:192]
DEST[191:160] := SRC2[223:192]
DEST[223:192] := SRC1[255:224]
DEST[255:224] := SRC2[255:224]
DEST[MAXVL-1:256] := 0
```

VUNPCKHPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

UNPCKHPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[127:96] := SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKHPS __m512 _mm512_unpackhi_ps( __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-52, “Type E4NF Class Exception Conditions.”

UNPCKLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 14 /r VUNPCKLPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.
EVEX.256.66.0F.W1 14 /r VUNPCKLPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.
EVEX.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves double precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the low double precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPD (EVEX Encoded Versions When SRC2 is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[127:64] := SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[255:192] := SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[383:320] := SRC2[319:256]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[511:448] := SRC2[447:384]
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPD (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[127:64] := TMP_SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] := SRC1[191:128]
    TMP_DEST[255:192] := TMP_SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] := SRC1[319:256]
    TMP_DEST[383:320] := TMP_SRC2[319:256]
    TMP_DEST[447:384] := SRC1[447:384]
    TMP_DEST[511:448] := TMP_SRC2[447:384]
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[191:128] := SRC1[191:128]
DEST[255:192] := SRC2[191:128]
DEST[MAXVL-1:256] := 0
```

VUNPCKLPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0
```

UNPCKLPD (128-bit Legacy SSE Version)

```
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPD __m512d _mm512_unpacklo_pd(__m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)
VUNPCKLPD __m256d _mm256_mask_unpacklo_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)
VUNPCKLPD __m128d _mm_mask_unpacklo_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLPD __m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-52, "Type E4NF Class Exception Conditions."

UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 14 /r UNPCKLPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.256.0F.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Unpacks and Interleaves single precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.512.0F.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Unpacks and Interleaves single precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an interleaved unpack of the low single precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

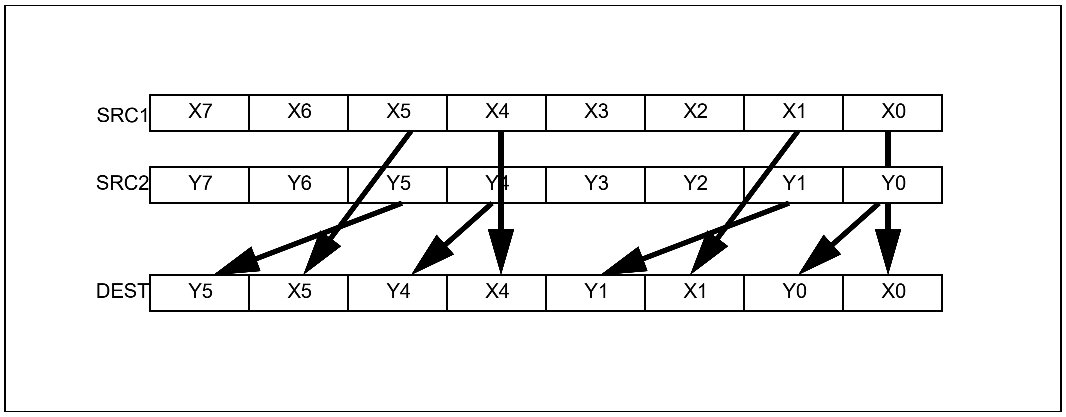


Figure 4-28. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPS (EVEX Encoded Version When SRC2 is a ZMM Register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] := SRC1[31:0]
    TMP_DEST[63:32] := SRC2[31:0]
    TMP_DEST[95:64] := SRC1[63:32]
    TMP_DEST[127:96] := SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[159:128]
    TMP_DEST[191:160] := SRC2[159:128]
    TMP_DEST[223:192] := SRC1[191:160]
    TMP_DEST[255:224] := SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[287:256]
    TMP_DEST[319:288] := SRC2[287:256]
    TMP_DEST[351:320] := SRC1[319:288]
    TMP_DEST[383:352] := SRC2[319:288]
    TMP_DEST[415:384] := SRC1[415:384]
    TMP_DEST[447:416] := SRC2[415:384]
    TMP_DEST[479:448] := SRC1[447:416]
    TMP_DEST[511:480] := SRC2[447:416]
FI;
FOR j := 0 TO KL-1
```

```
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VUNPCKLPS (EVEX Encoded Version When SRC2 is Memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
TMP_DEST[31:0] := SRC1[31:0]
TMP_DEST[63:32] := TMP_SRC2[31:0]
TMP_DEST[95:64] := SRC1[63:32]
TMP_DEST[127:96] := TMP_SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] := SRC1[159:128]
    TMP_DEST[191:160] := TMP_SRC2[159:128]
    TMP_DEST[223:192] := SRC1[191:160]
    TMP_DEST[255:224] := TMP_SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] := SRC1[287:256]
    TMP_DEST[319:288] := TMP_SRC2[287:256]
    TMP_DEST[351:320] := SRC1[319:288]
    TMP_DEST[383:352] := TMP_SRC2[319:288]
    TMP_DEST[415:384] := SRC1[415:384]
    TMP_DEST[447:416] := TMP_SRC2[415:384]
    TMP_DEST[479:448] := SRC1[447:416]
    TMP_DEST[511:480] := TMP_SRC2[447:416]
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

UNPCKLPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[159:128] := SRC1[159:128]
DEST[191:160] := SRC2[159:128]
DEST[223:192] := SRC1[191:160]
DEST[255:224] := SRC2[191:160]
DEST[MAXVL-1:256] := 0
```

VUNPCKLPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[MAXVL-1:128] := 0
```

UNPCKLPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[127:96] := SRC2[63:32]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKLPS __m512 _mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_mask_unpacklo_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 _mm_mask_unpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-52, “Type E4NF Class Exception Conditions.”

TZCNT—Count the Number of Trailing Zero Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F BC /r TZCNT r16, r/m16	A	V/V	BMI1	Count the number of trailing zero bits in r/m16, return result in r16.
F3 0F BC /r TZCNT r32, r/m32	A	V/V	BMI1	Count the number of trailing zero bits in r/m32, return result in r32.
F3 REX.W 0F BC /r TZCNT r64, r/m64	A	V/N.E.	BMI1	Count the number of trailing zero bits in r/m64, return result in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in the destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between the TZCNT and BSF instructions is that when the source operand is zero, TZCNT outputs the operand size to the destination operand, whereas BSF leaves the destination operand unmodified.

On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

```
temp := 0
DEST := 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )

    temp := temp +1
    DEST := DEST+ 1
OD

IF DEST = OperandSize
    CF := 1
ELSE
    CF := 0
FI

IF DEST = 0
    ZF := 1
ELSE
    ZF := 0
FI
```

Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF, and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```
TZCNT unsigned __int32 _tzcnt_u32(unsigned __int32 src);
TZCNT unsigned __int64 _tzcnt_u64(unsigned __int64 src);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#UD	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If LOCK prefix is used.

8. Updates to Chapter 5, Volume 2C

Change bars and violet text show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C:* Instruction Set Reference, V.

Changes to this chapter:

- Revised Description to provide return integer value for convert instructions:
  - VCVTPD2QQ
  - VCVTPD2UDQ
  - VCVTPD2UQQ
  - VCVTPH2DQ
  - VCVTPH2QQ
  - VCVTPH2UDQ
  - VCVTPH2UQQ
  - VCVTPH2UW
  - VCVTPH2W
  - VCVTPS2QQ
  - VCVTPS2UDQ
  - VCVTPS2UQQ
  - VCVTSD2USI
  - VCVTSH2SI
  - VCVTSH2USI
  - VCVTSS2USI
  - VCVTTPD2QQ
  - VCVTTPD2UDQ
  - VCVTTPD2UQQ
  - VCVTTPH2DQ
  - VCVTTPH2QQ
  - VCVTTPH2UDQ
  - VCVTTPH2UQQ
  - VCVTTPH2UW
  - VCVTTPH2W
  - VCVTTPS2QQ
  - VCVTTPS2UDQ
  - VCVTTPS2UQQ
  - VCVTTSD2USI
  - VCVTTSH2SI
  - VCVTTSH2USI
  - VCVTTSS2USI
- Corrected the exception type for EVEX-encoded instructions VCVTPS2QQ, VCVTPS2UQQ, VCVTTPS2QQ, and VCVTTPS2UQQ.
- Removed footnote references to verify vector options for the following instructions:
  - VADDPH
  - VADDSH
  - VALIGND/VALIGNQ
  - VBLENDMPD/VBLENDMPS
  - VBROADCAST

- VCMPPH
- VCMPSH
- VCOMISH
- VCOMPRESSPD
- VCOMPRESSPS
- VCVTDQ2PH
- VCVTNE2PS2BF16
- VCVTNEPS2BF16
- VCVTPD2PH
- VCVTPD2QQ
- VCVTPD2UDQ
- VCVTPD2UQQ
- VCVTPH2DQ
- VCVTPH2PD
- VCVTPH2PS/VCVTPH2PSX
- VCVTPH2QQ
- VCVTPH2UDQ
- VCVTPH2UQQ
- VCVTPH2UW
- VCVTPH2W
- VCVTPS2PH
- VCVTPS2PHX
- VCVTPS2QQ
- VCVTPS2UDQ
- VCVTPS2UQQ
- VCVTQQ2PD
- VCVTQQ2PH
- VCVTQQ2PS
- VCVTSD2SH
- VCVTSD2USI
- VCVTSH2SD
- VCVTSH2SI
- VCVTSH2SS
- VCVTSH2USI
- VCVTSI2SH
- VCVTSS2SH
- VCVTSS2USI
- VCVTTPD2QQ
- VCVTTPD2UDQ
- VCVTTPD2UQQ
- VCVTTPH2DQ
- VCVTTPH2QQ
- VCVTTPH2UDQ
- VCVTTPH2UQQ
- VCVTTPH2UW
- VCVTTPH2W

— VCVTTPS2QQ
— VCVTTPS2UDQ
— VCVTTPS2UQQ
— VCVTTSD2USI
— VCVTTSH2SI
— VCVTTSH2USI
— VCVTTSS2USI
— VCVTUDQ2PD
— VCVTUDQ2PH
— VCVTUDQ2PS
— VCVTUQQ2PD
— VCVTUQQ2PH
— VCVTUQQ2PS
— VCVTUSI2SD
— VCVTUSI2SS
— VCVTUW2PH
— VCVTW2PH
— VDBPSADBW
— VDIVPH
— VDIVSH
— VDPBF16PS
— VEXPANDPD
— VEXPANDPS
— VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4
— VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4
— VFCMADDCPH/VFMADDCPH
— VFCMADDCSH/VFMADDCSH
— VFCMULCPH/VFMULCPH
— VFCMULCSH/VFMULCSH
— VFIXUPIMMPD
— VFIXUPIMMPS
— VFIXUPIMMSD
— VFIXUPIMMSS
— VFMADD132PD/VFMADD213PD/VFMADD231PD
— VF[,N]MADD[132,213,231]PH
— VFMADD132PS/VFMADD213PS/VFMADD231PS
— VFMADD132SD/VFMADD213SD/VFMADD231SD
— VF[,N]MADD[132,213,231]SH
— VFMADD132SS/VFMADD213SS/VFMADD231SS
— VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD
— VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH
— VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS
— VFMSUB132PD/VFMSUB213PD/VFMSUB231PD
— VF[,N]MSUB[132,213,231]PH
— VFMSUB132PS/VFMSUB213PS/VFMSUB231PS
— VFMSUB132SD/VFMSUB213SD/VFMSUB231SD

— VF[,N]MSUB[132,213,231]SH
— VFMSUB132SS/VFMSUB213SS/VFMSUB231SS
— VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD
— VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH
— VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS
— VFNMADD132PD/VFNMADD213PD/VFNMADD231PD
— VFNMADD132PS/VFNMADD213PS/VFNMADD231PS
— VFNMADD132SD/VFNMADD213SD/VFNMADD231SD
— VFNMADD132SS/VFNMADD213SS/VFNMADD231SS
— VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD
— VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS
— VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD
— VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS
— VFPCLASSPD
— VFPCLASSPH
— VFPCLASSPS
— VFPCLASSSD
— VFPCLASSSH
— VFPCLASSSS
— VGATHERDPS/VGATHERDPD
— VGATHERQPS/VGATHERQPD
— VGETEXPPD
— VGETEXPPH
— VGETEXPPS
— VGETEXPSD
— VGETEXPSH
— VGETEXPSS
— VGETMANTPD
— VGETMANTPH
— VGETMANTPS
— VGETMANTSD
— VGETMANTSH
— VGETMANTSS
— VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4
— VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4
— VMAXPH
— VMAXSH
— VMINPH
— VMINSH
— VMOVSH
— VMOVW
— VMULPH
— VMULSH
— VPBLENDMB/VPBLENDMW
— VPBLENDMD/VPBLENDMQ
— VPBROADCAST

— VPBROADCASTB/W/D/Q
— VPBROADCASTM
— VPCMPB/VPCMPUB
— VPCMPD/VPCMPUD
— VPCMPQ/VPCMPUQ
— VPCMPW/VPCMPUW
— VPCOMPRESSB/VCOMPRESSW
— VPCOMPRESSD
— VPCOMPRESSQ
— VPCONFLICTD/Q
— VPDPBUSD
— VPDPBUSDS
— VPDPWSSD
— VPDPWSSDS
— VPERMB
— VPERMD/VPERMW
— VPERMI2B
— VPERMI2W/D/Q/PS/PD
— VPERMILPD
— VPERMILPS
— VPERMPD
— VPERMPS
— VPERMQ
— VPERMT2B
— VPERMT2W/D/Q/PS/PD
— VPEXPANDB/VPEXPANDW
— VPEXPANDD
— VPEXPANDQ
— VPGATHERDD/VPGATHERDQ
— VPGATHERQD/VPGATHERQQ
— VPLZCNTD/Q
— VPMADD52HUQ
— VPMADD52LUQ
— VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M
— VPMOVDB/VPMOVSDB/VPMOVUSDB
— VPMOVDW/VPMOVSDW/VPMOVUSDW
— VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q
— VPMOVQB/VPMOVSQB/VPMOVUSQB
— VPMOVQD/VPMOVSQD/VPMOVUSQD
— VPMOVQW/VPMOVSQW/VPMOVUSQW
— VPMOVWB/VPMOVSWB/VPMOVUSWB
— VPMULTISHIFTQB
— VPOPCNT
— VPROLD/VPROLVD/VPROLQ/VPROLVQ
— VPRORD/VPRORVD/VPRORQ/VPRORVQ
— VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ

- VPSHLD
- VPSHLDV
- VPSHRD
- VPSHRDV
- VPSHUFBITQMB
- VPSLLVW/VPSLLVD/VPSLLVQ
- VPSRAVW/VPSRAVD/VPSRAVQ
- VPSRLVW/VPSRLVD/VPSRLVQ
- VPTERNLOGD/VPTERNLOGQ
- VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ
- VPTESTNMB/W/D/Q
- VRANGEPD
- VRANGEPS
- VRANGESD
- VRANGESS
- VRCP14PD
- VRCP14PS
- VRCP14SD
- VRCP14SS
- VRCPPH
- VRCPSH
- VREDUCEPD
- VREDUCEPH
- VREDUCEPS
- VREDUCESD
- VREDUCESH
- VREDUCESS
- VRNDSCALEPD
- VRNDSCALEPH
- VRNDSCALEPS
- VRNDSCALESD
- VRNDSCALESH
- VRNDSCALESS
- VRSQRT14PD
- VRSQRT14PS
- VRSQRT14SD
- VRSQRT14SS
- VRSQRTPH
- VRSQRTSH
- VSCALEFPD
- VSCALEFPH
- VSCALEFPS
- VSCALEFSD
- VSCALEFSH
- VSCALEFSS
- VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD

- — VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2
- — VSQRTPH
- — VSQRTSH
- — VSUBPH
- — VSUBSH
- — VUCOMISH

CHAPTER 5
INSTRUCTION SET REFERENCE, V

5.1 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-1 for the column index value between 0:7, followed by Table 5-2 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a short-hand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one of more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator ‘not’ is expressed using ‘!’. Additionally, the conditional expression “A?B:C” expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g., andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g., andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g., norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g., zeros or norCB.

The 3-input expression “majorABC” returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression “minorABC” returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-1 and Table 5-2 include:

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.

Table 5-1. Lower 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	**[7:4]**							
[3:0]	*0H*	*1H*	*2H*	*3H*	*4H*	*5H*	*6H*	*7H*
00H	*FALSE*	*andAnorBC*	*norBnandAC*	*andA!B*	*norCnandBA*	*andA!C*	*andAxorBC*	*andAnandBC*
01H	*norABC*	*norCB*	*norBxorAC*	*A?!B:norBC*	*norCxorBA*	*A?!C:norBC*	*A?xorBC:norBC*	*A?nandBC:norBC*
02H	*andCnorBA*	*norBxnorAC*	*andC!B*	*norBnorAC*	*C?norBA:andBA*	*C?norBA:A*	*C?!B:andBA*	*C?!B:A*
03H	*norBA*	*norBandAC*	*C?!B:norBA*	*!B*	*C?norBA:xnorBA*	*A?!C:!B*	*A?xorBC:!B*	*A?nandBC:!B*
04H	*andBnorAC*	*norCxnorBA*	*B?norAC:andAC*	*B?norAC:A*	*andB!C*	*norCnorBA*	*B?!C:andAC*	*B?!C:A*
05H	*norCA*	*norCandBA*	*B?norAC:xnorAC*	*A?!B:!C*	*B?!C:norAC*	*!C*	*A?xorBC:!C*	*A?nandBC:!C*
06H	*norAxnorBC*	*A?norBC:xorBC*	*B?norAC:C*	*xorBorAC*	*C?norBA:B*	*xorCorBA*	*xorCB*	*B?!C:orAC*
07H	*norAandBC*	*minorABC*	*C?!B:!A*	*nandBorAC*	*B?!C:!A*	*nandCorBA*	*A?xorBC:nandBC*	*nandCB*
08H	*norAnandBC*	*A?norBC:andBC*	*andCxorBA*	*A?!B:andBC*	*andBxorAC*	*A?!C:andBC*	*A?xorBC:andBC*	*xorAandBC*
09H	*norAxorBC*	*A?norBC:xnorBC*	*C?xorBA:norBA*	*A?!B:xnorBC*	*B?xorAC:norAC*	*A?!C:xnorBC*	*xnorABC*	*A?nandBC:xnorBC*
0AH	*andC!A*	*A?norBC:C*	*andCnandBA*	*A?!B:C*	*C?!A:andBA*	*xorCA*	*xorCandBA*	*A?nandBC:C*
0BH	*C?!A:norBA*	*C?!A:!B*	*C?nandBA:norBA*	*C?nandBA:!B*	*B?xorAC:!A*	*B?xorAC:nandAC*	*C?nandBA:xnorBA*	*nandBxnorAC*
0CH	*andB!A*	*A?norBC:B*	*B?!A:andAC*	*xorBA*	*andBnandAC*	*A?!C:B*	*xorBandAC*	*A?nandBC:B*
0DH	*B?!A:norAC*	*B?!A:!C*	*B?!A:xnorAC*	*C?xorBA:nandBA*	*B?nandAC:norAC*	*B?nandAC:!C*	*B?nandAC:xnorAC*	*nandCxnorBA*
0EH	*norAnorBC*	*xorAorBC*	*B?!A:C*	*A?!B:orBC*	*C?!A:B*	*A?!C:orBC*	*B?nandAC:C*	*A?nandBC:orBC*
0FH	*!A*	*nandAorBC*	*C?nandBA:!A*	*nandBA*	*B?nandAC:!A*	*nandCA*	*nandAxnorBC*	*nandABC*

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.

Table 5-2. Upper 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	**[7:4]**							
[3:0]	*08H*	*09H*	*0AH*	*0BH*	*0CH*	*0DH*	*0EH*	*0FH*
00H	*andABC*	*andAxnorBC*	*andCA*	*B?andAC:A*	*andBA*	*C?andBA:A*	*andAorBC*	*A*
01H	*A?andBC:norBC*	*B?andAC:!C*	*A?C:norBC*	*C?A:!B*	*A?B:norBC*	*B?A:!C*	*xnorAorBC*	*orAnorBC*
02H	*andCxnorBA*	*B?andAC:xorAC*	*B?andAC:C*	*B?andAC:orAC*	*C?xnorBA:andBA*	*B?A:xorAC*	*B?A:C*	*B?A:orAC*
03H	*A?andBC:!B*	*xnorBandAC*	*A?C:!B*	*nandBnandAC*	*xnorBA*	*B?A:nandAC*	*A?orBC:!B*	*orA!B*
04H	*andBxnorAC*	*C?andBA:xorBA*	*B?xnorAC:andAC*	*B?xnorAC:A*	*C?andBA:B*	*C?andBA:orBA*	*C?A:B*	*C?A:orBA*
05H	*A?andBC:!C*	*xnorCandBA*	*xnorCA*	*C?A:nandBA*	*A?B:!C*	*nandCnandBA*	*A?orBC:!C*	*orA!C*
06H	*A?andBC:xorBC*	*xorABC*	*A?C:xorBC*	*B?xnorAC:orAC*	*A?B:xorBC*	*C?xnorBA:orBA*	*A?orBC:xorBC*	*orAxorBC*
07H	*xnorAandBC*	*A?xnorBC:nandBC*	*A?C:nandBC*	*nandBxorAC*	*A?B:nandBC*	*nandCxorBA*	*A?orBCnandBC*	*orAnandBC*
08H	*andCB*	*A?xnorBC:andBC*	*andCorAB*	*B?C:A*	*andBorAC*	*C?B:A*	*majorABC*	*orAandBC*
09H	*B?C:norAC*	*xnorCB*	*xnorCorBA*	*C?orBA:!B*	*xnorBorAC*	*B?orAC:!C*	*A?orBC:xnorBC*	*orAxnorBC*
0AH	*A?andBC:C*	*A?xnorBC:C*	*C*	*B?C:orAC*	*A?B:C*	*B?orAC:xorAC*	*orCandBA*	*orCA*
0BH	*B?C:!A*	*B?C:nandAC*	*orCnorBA*	*orC!B*	*B?orAC:!A*	*B?orAC:nandAC*	*orCxnorBA*	*nandBnorAC*
0CH	*A?andBC:B*	*A?xnorBC:B*	*A?C:B*	*C?orBA:xorBA*	*B*	*C?B:orBA*	*orBandAC*	*orBA*
0DH	*C?B!A*	*C?B:nandBA*	*C?orBA:!A*	*C?orBA:nandBA*	*orBnorAC*	*orB!C*	*orBxnorAC*	*nandCnorBA*
0EH	*A?andBC:orBC*	*A?xnorBC:orBC*	*A?C:orBC*	*orCxorBA*	*A?B:orBC*	*orBxorAC*	*orCB*	*orABC*
0FH	*nandAnandBC*	*nandAxorBC*	*orC!A*	*orCnandBA*	*orB!A*	*orBnandAC*	*nandAorBC*	*TRUE*

Table 5-1 and Table 5-2 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

Result_imm8 = Table_Lookup_Entry(0F0H, 0CCH, 0AAH)

Table_Lookup_Entry is the Boolean expression defined in Table 5-1 and Table 5-2.

5.2 INSTRUCTIONS (V)

Chapter 5 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (V). See also: Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A; Chapter 5, “Instruction Set Reference, V,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B; and Chapter 5, “Instruction Set Reference, V,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

VADDPH—Add Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 58 /r VADDPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Add packed FP16 value from xmm3/m128/m16bcst to xmm2, and store result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 58 /r VADDPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Add packed FP16 value from ymm3/m256/m16bcst to ymm2, and store result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 58 /r VADDPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Add packed FP16 value from zmm3/m512/m16bcst to zmm2, and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction adds packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Register

```
VL = 128, 256 or 512
KL := VL/16
IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSEIF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
VL = 128, 256 or 512
KL := VL/16
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDPH __m128h _mm_add_ph (__m128h a, __m128h b);
VADDPH __m128h _mm_mask_add_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDPH __m128h _mm_maskz_add_ph (__mmask8 k, __m128h a, __m128h b);
VADDPH __m256h _mm256_add_ph (__m256h a, __m256h b);
VADDPH __m256h _mm256_mask_add_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VADDPH __m256h _mm256_maskz_add_ph (__mmask16 k, __m256h a, __m256h b);
VADDPH __m512h _mm512_add_ph (__m512h a, __m512h b);
VADDPH __m512h _mm512_add_ph (__m512h a, __m512h b);
VADDPH __m512h _mm512_mask_add_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_maskz_add_ph (__mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_add_round_ph (__m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_mask_add_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_maskz_add_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VADDSH—Add Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 58 /r VADDSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Add the low FP16 value from xmm3/m16 to xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction adds the low FP16 value from the source operands and stores the FP16 result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VADDSH (EVEX Encoded Versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] + SRC2.fp16[0]
ELSEIF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged
DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VADDSH __m128h _mm_add_round_sh (__m128h a, __m128h b, int rounding);
VADDSH ___m128h _mm_mask_add_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH ___m128h _mm_maskz_add_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH ___m128h _mm_add_sh (__m128h a, __m128h b);
VADDSH ___m128h _mm_mask_add_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDSH ___m128h _mm_maskz_add_sh (__mmask8 k, __m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 03 /r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors xmm2 and xmm3/m128/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.128.66.0F3A.W1 03 /r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors xmm2 and xmm3/m128/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.256.66.0F3A.W0 03 /r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors ymm2 and ymm3/m256/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.256.66.0F3A.W1 03 /r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift right and merge vectors ymm2 and ymm3/m256/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.512.66.0F3A.W0 03 /r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shift right and merge vectors zmm2 and zmm3/m512/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.
EVEX.512.66.0F3A.W1 03 /r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shift right and merge vectors zmm2 and zmm3/m512/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).

Operation

VALIGND (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j := 0 TO KL-1
            i := j * 32
            src[i+31:i] := SRC2[31:0]
        ENDFOR;
    ELSE src := SRC2
FI
; Concatenate sources
tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
    THEN SHIFT = imm8[1:0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[2:0]
            ELSE SHIFT = imm8[3:0]
        FI
FI;
tmp[2VL-1:0] := tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := tmp[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

VALIGNQ (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256),(8, 512)
IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j := 0 TO KL-1
            i := j * 64
            src[i+63:i] := SRC2[63:0]
        ENDFOR;
    ELSE src := SRC2
FI
; Concatenate sources
tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right quadword elements
```

```
IF VL = 128
    THEN SHIFT = imm8[0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[1:0]
            ELSE SHIFT = imm8[2:0]
        FI
FI;
tmp[2VL-1:0] := tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := tmp[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VALIGND __m512i _mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i _mm256_mask_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i _mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i _mm_mask_alignr_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i _mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i _mm512_alignr_epi64( __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64( __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i _mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i _mm256_maskz_alignr_epi64( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i _mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i _mm_maskz_alignr_epi64( __mmask8 k, __m128i a, __m128i b, int cnt);
```

Exceptions

See Table 2-52, "Type E4NF Class Exception Conditions."

VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 65 /r VBLENDMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend double precision vector xmm2 and double precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W1 65 /r VBLENDMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend double precision vector ymm2 and double precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W1 65 /r VBLENDMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Blend double precision vector zmm2 and double precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.
EVEX.128.66.0F38.W0 65 /r VBLENDMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend single precision vector xmm2 and single precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W0 65 /r VBLENDMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend single precision vector ymm2 and single precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W0 65 /r VBLENDMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Blend single precision vector zmm2 and single precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

Operation

VBLENDMPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC2[63:0]
                ELSE
                    DEST[i+63:i] := SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+63:i] := SRC1[i+63:i]
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBLENDMPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 _mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	V/V	AVX	Broadcast single precision floating-point element in mem to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32	A	V/V	AVX	Broadcast single precision floating-point element in mem to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64	A	V/V	AVX	Broadcast double precision floating-point element in mem to four locations in ymm1.
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128	A	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
VEX.128.66.0F38.W0 18/r VBROADCASTSS xmm1, xmm2	A	V/V	AVX2	Broadcast the low single precision floating-point element in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, xmm2	A	V/V	AVX2	Broadcast low single precision floating-point element in the source operand to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, xmm2	A	V/V	AVX2	Broadcast low double precision floating-point element in the source operand to four locations in ymm1.
EVEX.256.66.0F38.W1 19 /r VBROADCASTSD ymm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast low double precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F OR AVX10.1	Broadcast low double precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast two single precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ OR AVX10.1	Broadcast two single precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast low single precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast low single precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F OR AVX10.1	Broadcast low single precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 {k1}{z}, m128	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast 128 bits of 4 single precision floating-point data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F OR AVX10.1	Broadcast 128 bits of 4 single precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 {k1}{z}, m128	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast 128 bits of 2 double precision floating-point data in mem to locations in ymm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ OR AVX10.1	Broadcast 128 bits of 2 double precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 1B /r VBROADCASTF32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ OR AVX10.1	Broadcast 256 bits of 8 single precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 1B /r VBROADCASTF64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F OR AVX10.1	Broadcast 256 bits of 4 double precision floating-point data in mem to locations in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

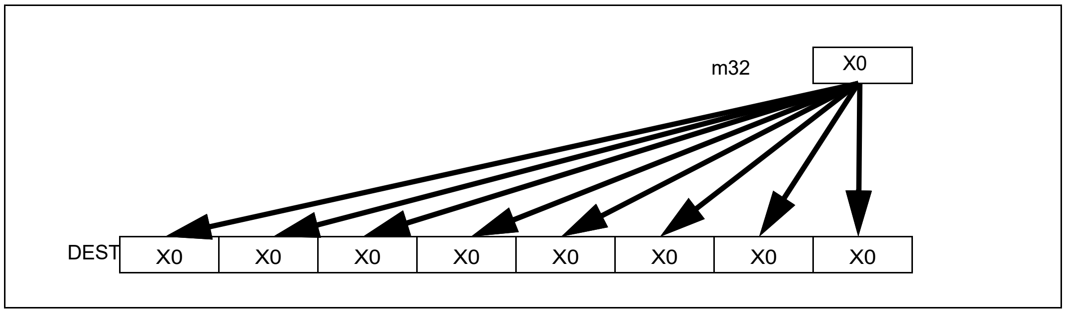


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

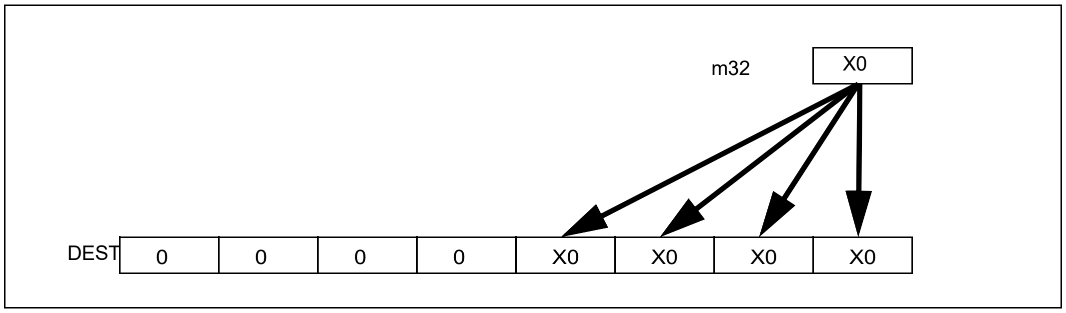


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

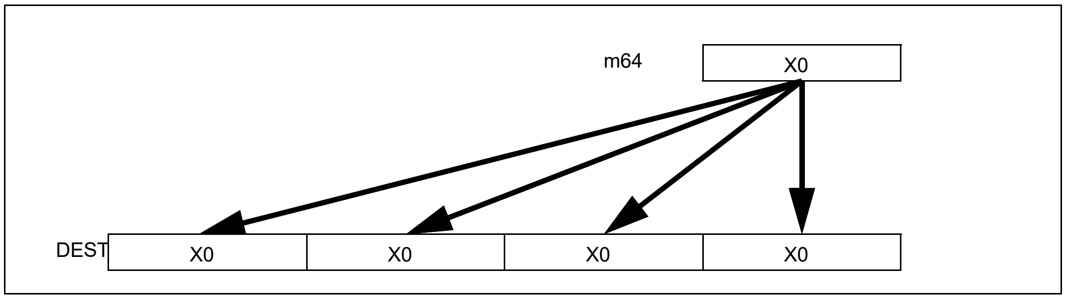


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

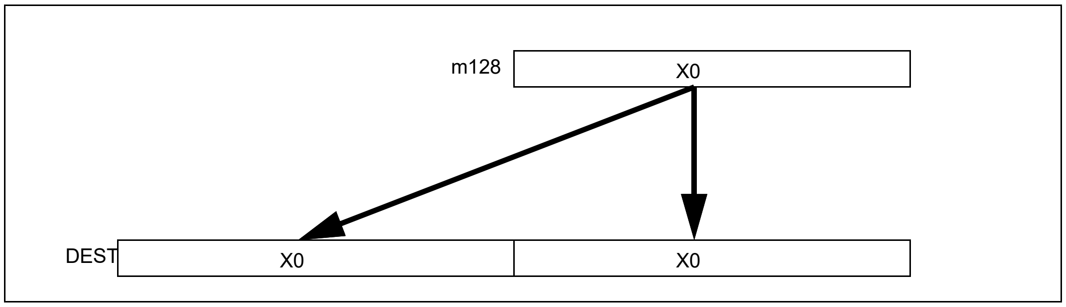


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)

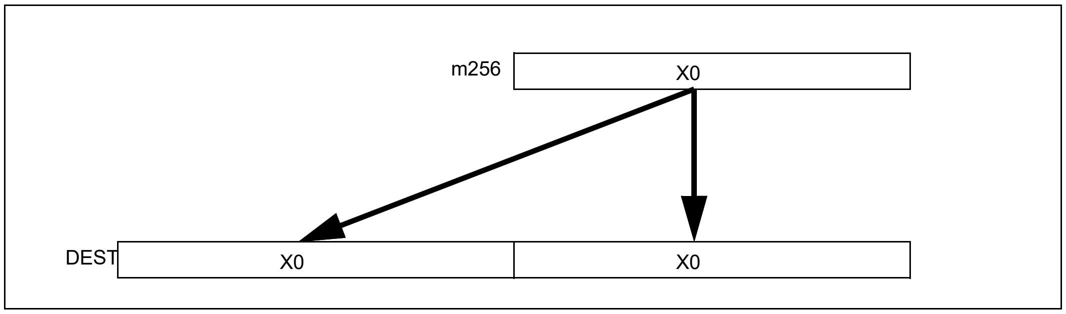


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

Operation

VBROADCASTSS (128-bit Version VEX and Legacy)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[MAXVL-1:128] := 0
```

VBROADCASTSS (VEX.256 Encoded Version)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[159:128] := temp
DEST[191:160] := temp
DEST[223:192] := temp
DEST[255:224] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTSS (EVEX Encoded Versions)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTSD (VEX.256 Encoded Version)

```
temp := SRC[63:0]
DEST[63:0] := temp
DEST[127:64] := temp
DEST[191:128] := temp
DEST[255:192] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTSD (EVEX Encoded Versions)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF32x2 (EVEX Encoded Versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    n := (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF128 (VEX.256 Encoded Version)

```
temp := SRC[127:0]
DEST[127:0] := temp
DEST[255:128] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTF32X4 (EVEX Encoded Versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j* 32
    n := (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF64X2 (EVEX Encoded Versions)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    n := (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR;
```

VBROADCASTF32X8 (EVEX.U1.512 Encoded Version)

```
FOR j := 0 TO 15
    i := j * 32
    n := (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTF64X4 (EVEX.512 Encoded Version)

```
FOR j := 0 TO 7
    i := j * 64
    n := (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2( __mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m256 _mm256_mask_broadcast_f32x2(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_maskz_broadcast_f32x2( __mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4( __mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m256 _mm256_mask_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_maskz_broadcast_f32x4( __mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8( __m256 a);
VBROADCASTF32x8 __m512 _mm512_mask_broadcast_f32x8(__m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 _mm512_maskz_broadcast_f32x8( __mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d _mm512_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m512d _mm512_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m256d _mm256_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512_broadcast_f64x4( __m256d a);
VBROADCASTF64x4 __m512d _mm512_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d _mm512_maskz_broadcast_f64x4( __mmask8 k, __m256d a);
VBROADCASTSD __m512d _mm512_broadcastsd_pd( __m128d a);
VBROADCASTSD __m512d _mm512_mask_broadcastsd_pd(__m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcastsd_pd(__m128d a);
VBROADCASTSD __m256d _mm256_mask_broadcastsd_pd(__m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 _mm512_broadcastss_ps( __m128 a);
VBROADCASTSS __m512 _mm512_mask_broadcastss_ps(__m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 _mm512_maskz_broadcastss_ps( __mmask16 k, __m128 a);
VBROADCASTSS __m256 _mm256_broadcastss_ps(__m128 a);
VBROADCASTSS __m256 _mm256_mask_broadcastss_ps(__m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 _mm256_maskz_broadcastss_ps( __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 _mm_mask_broadcastss_ps(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_maskz_broadcastss_ps( __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);
```

Exceptions

VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
	If EVEX.L’L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.
	If EVEX.L’L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

VCMPPH—Compare Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, xmm2, xmm3/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compare packed FP16 values in xmm3/m128/m16bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.
EVEX.256.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, ymm2, ymm3/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compare packed FP16 values in ymm3/m256/m16bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.
EVEX.512.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, zmm2, zmm3/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Compare packed FP16 values in zmm3/m512/m16bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction compares packed FP16 values from source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on each of the pairs of packed values. The destination elements are updated according to the writemask.

Operation

```
CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
```

```
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
31: CMP_OPERATOR := TRUE_US;
ESAC
```

VCMPPH (EVEX Encoded Versions)

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k2[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.bit[j] := SRC1.fp16[j] CMP_OPERATOR tsrc2
    ELSE
        DEST.bit[j] := 0

DEST[MAXKL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPPH ___mmask8 _mm_cmp_ph_mask (__m128h a, __m128h b, const int imm8);
VCMPPH ___mmask8 _mm_mask_cmp_ph_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);
VCMPPH ___mmask16 _mm256_cmp_ph_mask (__m256h a, __m256h b, const int imm8);
VCMPPH ___mmask16 _mm256_mask_cmp_ph_mask (__mmask16 k1, __m256h a, __m256h b, const int imm8);
VCMPPH ___mmask32 _mm512_cmp_ph_mask (__m512h a, __m512h b, const int imm8);
VCMPPH ___mmask32 _mm512_mask_cmp_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8);
VCMPPH ___mmask32 _mm512_cmp_round_ph_mask (__m512h a, __m512h b, const int imm8, const int sae);
VCMPPH ___mmask32 _mm512_mask_cmp_round_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8, const int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCMPSH—Compare Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F3A.W0 C2 /r /ib VCMPSH k1{k2}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Compare low FP16 values in xmm3/m16 and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction compares the FP16 values from the lowest element of the source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on the pair of packed FP16 values. The low destination bit is updated according to the write-mask. Bits MAXKL-1:1 of the destination operand are zeroed.

Operation

```
CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
```

```
31: CMP_OPERATOR := TRUE_US;
ESAC
```

VCMPSH (EVEX Encoded Versions)

```
IF k2[0] OR *no writemask*:
    DEST.bit[0] := SRC1.fp16[0] CMP_OPERATOR SRC2.fp16[0]
ELSE
    DEST.bit[0] := 0

DEST[MAXKL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPSH __mmask8 _mm_cmp_round_sh_mask (__m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __mmask8 _mm_mask_cmp_round_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __mmask8 _mm_cmp_sh_mask (__m128h a, __m128h b, const int imm8);
VCMPSH __mmask8 _mm_mask_cmp_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 2F /r VCOMISH xmm1, xmm2/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Compare low FP16 values in xmm1 and xmm2/m16, and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A

Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VCOMISH instruction differs from the VUCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The VUCOMISH instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCOMISH SRC1, SRC2

```
RESULT := OrderedCompare(SRC1.fp16[0],SRC2.fp16[0])
IF RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
ELSE IF RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
ELSE IF RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
ELSE: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMISH int _mm_comi_round_sh (__m128h a, __m128h b, const int imm8, const int sae);
VCOMISH int _mm_comi_sh (__m128h a, __m128h b, const int imm8);
VCOMISH int _mm_comieq_sh (__m128h a, __m128h b);
VCOMISH int _mm_comige_sh (__m128h a, __m128h b);
VCOMISH int _mm_comigt_sh (__m128h a, __m128h b);
VCOMISH int _mm_comile_sh (__m128h a, __m128h b);
VCOMISH int _mm_comilt_sh (__m128h a, __m128h b);
VCOMISH int _mm_comineq_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCOMPRESSPD—Store Sparse Packed Double Precision Floating-Point Values Into Dense Memory

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed double precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed double precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed double precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (store) up to 8 double precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand) The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VCOMPRESSPD (EVEX Encoded Versions) Store Form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENDFOR
```

VCOMPRESSPD (EVEX Encoded Versions) Reg-Reg Form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMPRESSPD __m512d _mm512_mask_compress_pd( __m512d s, __mmask8 k, __m512d a);
VCOMPRESSPD __m512d _mm512_maskz_compress_pd( __mmask8 k, __m512d a);
VCOMPRESSPD void _mm512_mask_compressstoreu_pd( void * d, __mmask8 k, __m512d a);
VCOMPRESSPD __m256d _mm256_mask_compress_pd( __m256d s, __mmask8 k, __m256d a);
VCOMPRESSPD __m256d _mm256_maskz_compress_pd( __mmask8 k, __m256d a);
VCOMPRESSPD void _mm256_mask_compressstoreu_pd( void * d, __mmask8 k, __m256d a);
VCOMPRESSPD __m128d _mm_mask_compress_pd( __m128d s, __mmask8 k, __m128d a);
VCOMPRESSPD __m128d _mm_maskz_compress_pd( __mmask8 k, __m128d a);
VCOMPRESSPD void _mm_mask_compressstoreu_pd( void * d, __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCOMPRESSPS—Store Sparse Packed Single Precision Floating-Point Values Into Dense Memory

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8A /r VCOMPRESSPS xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed single precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W0 8A /r VCOMPRESSPS ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed single precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed single precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 16 single precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VCOMPRESSPS (EVEX Encoded Versions) Store Form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR;
```

VCOMPRESSPS (EVEX Encoded Versions) Reg-Reg Form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
    ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMPRESSPS __m512 _mm512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 _mm512_maskz_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void _mm512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 _mm256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 _mm256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void _mm256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 _mm_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 _mm_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void _mm_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb. in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTDQ2PH—Convert Packed Signed Doubleword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5B /r VCVTDQ2PH xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5B /r VCVTDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5B /r VCVTDQ2PH ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts four, eight, or sixteen packed signed doubleword integers in the source operand to four, eight, or sixteen packed FP16 values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 32-bit memory location. The destination operand is a YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

```
VCVTDQ2PH DEST, SRC
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTDQ2PH __m256h _mm512_cvt_roundepi32_ph (__m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_mask_cvt_roundepi32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_maskz_cvt_roundepi32_ph (__mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m128h _mm_cvtepi32_ph (__m128i a);
VCVTDQ2PH __m128h _mm_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm_maskz_cvtepi32_ph (__mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm256_cvtepi32_ph (__m256i a);
VCVTDQ2PH __m128h _mm256_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTDQ2PH __m128h _mm256_maskz_cvtepi32_ph (__mmask8 k, __m256i a);
VCVTDQ2PH __m256h _mm512_cvtepi32_ph (__m512i a);
VCVTDQ2PH __m256h _mm512_mask_cvtepi32_ph (__m256h src, __mmask16 k, __m512i a);
VCVTDQ2PH __m256h _mm512_maskz_cvtepi32_ph (__mmask16 k, __m512i a);
```

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTNE2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F38.W0 72 /r VCVTNE2PS2BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from xmm2 and xmm3/m128/m32bcst to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F2.0F38.W0 72 /r VCVTNE2PS2BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from ymm2 and ymm3/m256/m32bcst to packed BF16 data in ymm1 with writemask k1.
EVEX.512.F2.0F38.W0 72 /r VCVTNE2PS2BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	(AVX512-BF16 AND AVX512F) OR AVX10.1	Convert packed single data from zmm2 and zmm3/m512/m32bcst to packed BF16 data in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts two SIMD registers of packed single data into a single register of packed BF16 data.

This instruction does not support memory fault suppression.

This instruction uses "Round to nearest (even)" rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated. No floating-point exceptions are generated.

Operation

VCVTNE2PS2BF16 dest, src1, src2

```
VL = (128, 256, 512)
KL = VL/16

origdest := dest
FOR i := 0 to KL-1:
    IF k1[ i ] or *no writemask*:
        IF i < KL/2:
            IF src2 is memory and evex.b == 1:
                t := src2.fp32[0]
            ELSE:
                t := src2.fp32[ i ]
        ELSE:
            t := src1.fp32[ i-KL/2]

        // See VCVTNEPS2BF16 for definition of convert helper function
        dest.word[i] := convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] := 0
    ELSE:  // Merge masking, dest element unchanged
        dest.word[ i ] := origdest.word[ i ]
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNE2PS2BF16 __m128bh _mm_cvtne2ps_pbh (__m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_mask_cvtne2ps_pbh (__m128bh, __mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_maskz_cvtne2ps_pbh (__mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m256bh _mm256_cvtne2ps_pbh (__m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_mask_cvtne2ps_pbh (__m256bh, __mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_maskz_cvtne2ps_ pbh (__mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m512bh _mm512_cvtne2ps_pbh (__m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_mask_cvtne2ps_pbh (__m512bh, __mmask32, __m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_maskz_cvtne2ps_pbh (__mmask32, __m512, __m512);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-52, “Type E4NF Class Exception Conditions.”

VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1, xmm2/m128	A	V/V	AVX-NE-CONVERT	Convert packed single precision floating-point values from xmm2/m128 to packed BF16 values and store in xmm1.
VEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1, ymm2/m256	A	V/V	AVX-NE-CONVERT	Convert packed single precision floating-point values from ymm2/m256 to packed BF16 values and store in xmm1.
EVEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512-BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512-BF16 AND AVX512VL) OR AVX10.1	Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1.
EVEX.512.F3.0F38.W0 72 /r VCVTNEPS2BF16 ymm1{k1}{z}, zmm2/m512/m32bcst	B	V/V	(AVX512-BF16 AND AVX512F) OR AVX10.1	Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction loads packed FP32 elements from a SIMD register or memory, converts the elements to BF16, and writes the result to the destination SIMD register.

The upper bits of the destination register beyond the down-converted BF16 elements are zeroed.

This instruction uses "Round to nearest (even)" rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

As the instruction operand encoding table shows, the EVEX.vvvv field is not used for encoding an operand. EVEX.vvvv is reserved and must be 0b1111 otherwise instructions will #UD.

Operation

```
Define convert_fp32_to_bfloat16(x):
    IF x is zero or denormal:
        dest[15] := x[31] // sign preserving zero (denormal go to zero)
        dest[14:0] := 0
    ELSE IF x is infinity:
        dest[15:0] := x[31:16]
    ELSE IF x is NAN:
        dest[15:0] := x[31:16] // truncate and set MSB of the mantissa to force QNAN
        dest[6] := 1
    ELSE // normal number
        LSB := x[16]
        rounding_bias := 0x00007FFF + LSB
        temp[31:0] := x[31:0] + rounding_bias // integer add
        dest[15:0] := temp[31:16]
    RETURN dest
```

VCVTNEPS2BF16 dest, src (VEX encoded version)

```
VL = (128, 256)
KL = VL/16

FOR i := 0 to KL/2-1:
    t := src.fp32[i]
    dest.word[i] := convert_fp32_to_bfloat16(t)

DEST[MAXVL-1:VL/2] := 0
```

VCVTNEPS2BF16 dest, src (EVEX encoded version)

```
VL = (128, 256, 512)
KL = VL/16

origdest := dest
FOR i := 0 to KL/2-1:
    IF k1[ i ] or *no writemask*:
        IF src is memory and evex.b == 1:
            t := src.fp32[0]
        ELSE:
            t := src.fp32[ i ]

        dest.word[i] := convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] := 0
    ELSE:  // Merge masking, dest element unchanged
        dest.word[ i ] := origdest.word[ i ]
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTNEPS2BF16 __m128bh _mm_cvtneps_avx_pbh (__m128 __A);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_avx_pbh (__m256 __A);
VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128 a);
VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128 __A);
VCVTNEPS2BF16 __m128bh _mm_mask_cvtneps_pbh (__m128bh src, __mmask8 k, __m128 a);
VCVTNEPS2BF16 __m128bh _mm_maskz_cvtneps_pbh (__mmask8 k, __m128 a);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_pbh (__m256 a);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_pbh (__m256 __A);
VCVTNEPS2BF16 __m128bh _mm256_mask_cvtneps_pbh (__m128bh src, __mmask8 k, __m256 a);
VCVTNEPS2BF16 __m128bh _mm256_maskz_cvtneps_pbh (__mmask8 k, __m256 a);
VCVTNEPS2BF16 __m256bh _mm512_cvtneps_pbh (__m512 a);
VCVTNEPS2BF16 __m256bh _mm512_mask_cvtneps_pbh (__m256bh src, __mmask16 k, __m512 a);
VCVTNEPS2BF16 __m256bh _mm512_maskz_cvtneps_pbh (__mmask16 k, __m512 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VCVTPD2PH—Convert Packed Double Precision FP Values to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts two, four, or eight packed double precision floating-point values in the source operand (second operand) to two, four, or eight packed FP16 values in the destination operand (first operand). When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasts from a 64-bit memory location. The destination operand is a XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:128/64/32) of the corresponding destination are zeroed.

EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

This instruction uses MXCSR.DAZ for handling FP64 inputs. FP16 outputs can be normal or denormal, and are not conditionally flushed to zero.

Operation

VCVTPD2PH DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.double[0]
        ELSE
            tsrc := SRC.double[j]
        DEST.fp16[j] := Convert_fp64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2PH __m128h _mm512_cvt_roundpd_ph (__m512d a, int rounding);
VCVTPD2PH __m128h _mm512_mask_cvt_roundpd_ph (__m128h src, __mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm512_maskz_cvt_roundpd_ph (__mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm_cvtpd_ph (__m128d a);
VCVTPD2PH __m128h _mm_mask_cvtpd_ph (__m128h src, __mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm_maskz_cvtpd_ph (__mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm256_cvtpd_ph (__m256d a);
VCVTPD2PH __m128h _mm256_mask_cvtpd_ph (__m128h src, __mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm256_maskz_cvtpd_ph (__mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm512_cvtpd_ph (__m512d a);
VCVTPD2PH __m128h _mm512_mask_cvtpd_ph (__m128h src, __mmask8 k, __m512d a);
VCVTPD2PH __m128h _mm512_maskz_cvtpd_ph (__mmask8 k, __m512d a);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPD2QQ—Convert Packed Double Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from xmm2/m128/m64bcst to two packed signed quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed signed quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 7B /r VCVTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/m512/m64bcst to eight packed signed quadword integers in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed quad-word integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=         Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2QQ __m512i _mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i _mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i _mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i _mm512_maskz_cvt_roundpd_epi64( __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i _mm256_mask_cvtpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_maskz_cvtpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m128i _mm_mask_cvtpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i _mm_maskz_cvtpd_epi64( __mmask8 k, __m128d a);
VCVTPD2QQ __m256i _mm256_cvtpd_epi64 (__m256d src)
VCVTPD2QQ __m128i _mm_cvtpd_epi64 (__m128d src)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPD2UDQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.256.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.512.0F.W1 79 /r VCVTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
            THEN
                IF (EVEX.b = 1)
                    THEN
                        DEST[i+31:i] :=
                Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])
                    ELSE
                        DEST[i+31:i] :=
                Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
                FI;
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2UDQ __m256i _mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_maskz_cvt_roundpd_epu32( __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPD2UQQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert fourth packed double precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 79 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2UQQ __m512i _mm512_cvtpd_epu64( __m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_epu64( __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvt_roundpd_epu64( __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_maskz_cvt_roundpd_epu64( __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64( __mmask8 k, __m256d a);
VCVTPD2UQQ __m128i _mm_mask_cvtpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i _mm_maskz_cvtpd_epu64( __mmask8 k, __m128d a);
VCVTPD2UQQ __m256i _mm256_cvtpd_epu64 (__m256d src)
VCVTPD2UQQ __m128i _mm_cvtpd_epu64 (__m128d src)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPH2DQ—Convert Packed FP16 Values to Signed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 5B /r VCVTPH2DQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 5B /r VCVTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 5B /r VCVTPH2DQ zmm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed doubleword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

The destination elements are updated according to the writemask.

Operation

```
VCVTPH2DQ DEST, SRC
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.dword[j] := Convert_fp16_to_integer32(tsrc)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPH2DQ __m512i _mm512_cvt_roundph_epi32 (__m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_mask_cvt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_maskz_cvt_roundph_epi32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m128i _mm_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m128i _mm_mask_cvtph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2DQ __m128i _mm_maskz_cvtph_epi32 (__mmask8 k, __m128h a);
VCVTPH2DQ __m256i _mm256_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m256i _mm256_mask_cvtph_epi32 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2DQ __m256i _mm256_maskz_cvtph_epi32 (__mmask8 k, __m128h a);
VCVTPH2DQ __m512i _mm512_cvtph_epi32 (__m256h a);
VCVTPH2DQ __m512i _mm512_mask_cvtph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2DQ __m512i _mm512_maskz_cvtph_epi32 (__mmask16 k, __m256h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2PD—Convert Packed FP16 Values to FP64 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5A /r VCVTPH2PD xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m32/m16bcst to FP64 values, and store result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5A /r VCVTPH2PD ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m64/m16bcst to FP64 values, and store result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5A /r VCVTPH2PD zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert packed FP16 values in xmm2/m128/m16bcst to FP64 values, and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values to FP64 values in the destination register. The destination elements are updated according to the writemask.

This instruction handles both normal and denormal FP16 inputs.

Operation

VCVTPH2PD DEST, SRC

```
VL = 128, 256, or 512
KL := VL/64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp64[j] := Convert_fp16_to_fp64(tsrc)
    ELSE IF *zeroing*:
        DEST.fp64[j] := 0
    // else dest.fp64[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2PD __m512d _mm512_cvt_roundph_pd (__m128h a, int sae);
VCVTPH2PD __m512d _mm512_mask_cvt_roundph_pd (__m512d src, __mmask8 k, __m128h a, int sae);
VCVTPH2PD __m512d _mm512_maskz_cvt_roundph_pd (__mmask8 k, __m128h a, int sae);
VCVTPH2PD __m128d _mm_cvtph_pd (__m128h a);
VCVTPH2PD __m128d _mm_mask_cvtph_pd (__m128d src, __mmask8 k, __m128h a);
VCVTPH2PD __m128d _mm_maskz_cvtph_pd (__mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_cvtph_pd (__m128h a);
VCVTPH2PD __m256d _mm256_mask_cvtph_pd (__m256d src, __mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_maskz_cvtph_pd (__mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_cvtph_pd (__m128h a);
VCVTPH2PD __m512d _mm512_mask_cvtph_pd (__m512d src, __mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_maskz_cvtph_pd (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2PS/VCVTPH2PSX—Convert Packed FP16 Values to Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64	A	V/V	F16C	Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point value in xmm1.
VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128	A	V/V	F16C	Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point value in ymm1.
EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point values in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}{z}, xmm2/m128	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point values in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}{z}, ymm2/m256 {sae}	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256 to packed single precision floating-point values in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 13 /r VCVTPH2PSX xmm1{k1}{z}, xmm2/m64/m16bcst	C	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four packed single precision floating-point values, and store result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 13 /r VCVTPH2PSX ymm1{k1}{z}, xmm2/m128/m16bcst	C	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight packed single precision floating-point values, and store result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 13 /r VCVTPH2PSX zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	C	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen packed single precision floating-point values, and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) floating-point values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

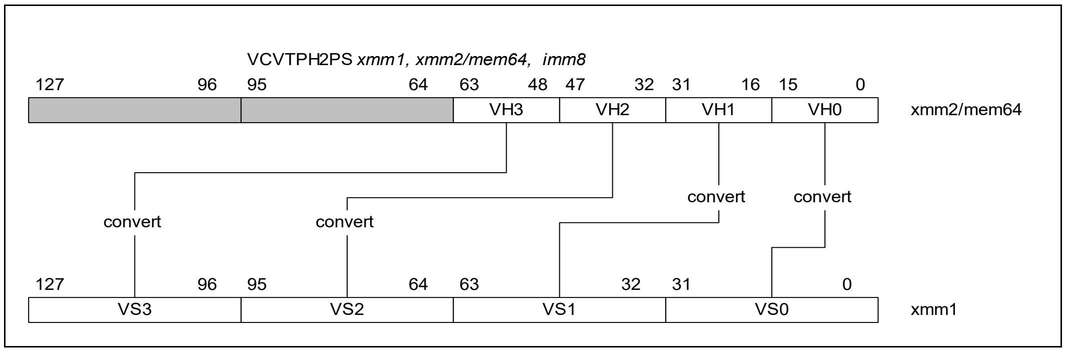


Figure 5-6. VCVTPH2PS (128-bit Version)

The VCVTPH2PSX instruction is a new form of the PH to PS conversion instruction, encoded in map 6. The previous version of the instruction, VCVTPH2PS, that is present in AVX512F (encoded in map 2, 0F38) does not support embedded broadcasting. The VCVTPH2PSX instruction has the embedded broadcasting option available.

The instructions associated with AVX512-FP16 always handle FP16 denormal number inputs; denormal inputs are not treated as zero.

Operation

```
vCvt_h2s(SRC1[15:0])
{
RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}
```

VCVTPH2PS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            vCvt_h2s(SRC[k+15:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPH2PS (VEX.256 Encoded Version)
```
DEST[31:0] := vCvt_h2s(SRC1[15:0]);
DEST[63:32] := vCvt_h2s(SRC1[31:16]);
DEST[95:64] := vCvt_h2s(SRC1[47:32]);
DEST[127:96] := vCvt_h2s(SRC1[63:48]);
DEST[159:128] := vCvt_h2s(SRC1[79:64]);
DEST[191:160] := vCvt_h2s(SRC1[95:80]);
DEST[223:192] := vCvt_h2s(SRC1[111:96]);
DEST[255:224] := vCvt_h2s(SRC1[127:112]);
DEST[MAXVL-1:256] := 0
```

VCVTPH2PS (VEX.128 Encoded Version)
```
DEST[31:0] := vCvt_h2s(SRC1[15:0]);
DEST[63:32] := vCvt_h2s(SRC1[31:16]);
DEST[95:64] := vCvt_h2s(SRC1[47:32]);
DEST[127:96] := vCvt_h2s(SRC1[63:48]);
DEST[MAXVL-1:128] := 0
```

VCVTPH2PSX DEST, SRC
```
VL = 128, 256, or 512
KL := VL/32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp32[j] := Convert_fp16_to_fp32(tsrc)
    ELSE IF *zeroing*:
        DEST.fp32[j] := 0
    // else dest.fp32[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPH2PS __m512 _mm512_cvtph_ps( __m256i a);
VCVTPH2PS __m512 _mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_cvt_roundph_ps( __m256i a, int sae);
VCVTPH2PS __m512 _mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512 _mm512_maskz_cvt_roundph_ps( __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m256 _mm256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256 _mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_cvtph_ps ( __m128i m1);
VCVTPH2PS __m256 _mm256_cvtph_ps ( __m128i m1)
```

VCVTPH2PSX __m512 _mm512_cvtx_roundph_ps (__m256h a, int sae);
VCVTPH2PSX __m512 _mm512_mask_cvtx_roundph_ps (__m512 src, __mmask16 k, __m256h a, int sae);
VCVTPH2PSX __m512 _mm512_maskz_cvtx_roundph_ps (__mmask16 k, __m256h a, int sae);
VCVTPH2PSX __m128 _mm_cvtxph_ps (__m128h a);
VCVTPH2PSX __m128 _mm_mask_cvtxph_ps (__m128 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m128 _mm_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m256 _mm256_cvtxph_ps (__m128h a);
VCVTPH2PSX __m256 _mm256_mask_cvtxph_ps (__m256 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m256 _mm256_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m512 _mm512_cvtxph_ps (__m256h a);
VCVTPH2PSX __m512 _mm512_mask_cvtxph_ps (__m512 src, __mmask16 k, __m256h a);
VCVTPH2PSX __m512 _mm512_maskz_cvtxph_ps (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

VEX-encoded instructions: Invalid.

EVEX-encoded instructions: Invalid.
EVEX-encoded instructions with broadcast (VCVTPH2PSX): Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-26, "Type 11 Class Exception Conditions" (do not report #AC).

EVEX-encoded instructions, see Table 2-62, "Type E11 Class Exception Conditions."

EVEX-encoded instructions with broadcast (VCVTPH2PSX), see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7B /r VCVTPH2QQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 7B /r VCVTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 7B /r VCVTPH2QQ zmm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed quadword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

The destination elements are updated according to the writemask.

Operation

```
VCVTPH2QQ DEST, SRC
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_integer64(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPH2QQ __m512i _mm512_cvt_roundph_epi64 (__m128h a, int rounding);
VCVTPH2QQ __m512i _mm512_mask_cvt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2QQ __m512i _mm512_maskz_cvt_roundph_epi64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2QQ __m128i _mm_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m128i _mm_mask_cvtph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m128i _mm_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ __m256i _mm256_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m256i _mm256_mask_cvtph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m256i _mm256_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ __m512i _mm512_cvtph_epi64 (__m128h a);
VCVTPH2QQ __m512i _mm512_mask_cvtph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2QQ __m512i _mm512_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2UDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 79 /r VCVTPH2UDQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned doubleword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 79 /r VCVTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned doubleword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 79 /r VCVTPH2UDQ zmm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned doubleword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned doubleword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2UDQ DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
            DEST.dword[j] := Convert_fp16_to_unsigned_integer32(tsrc)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPH2UDQ __m512i _mm512_cvt_roundph_epu32 (__m256h a, int rounding);
VCVTPH2UDQ __m512i _mm512_mask_cvt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m512i _mm512_maskz_cvt_roundph_epu32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m128i _mm_cvtph_epu32 (__m128h a);
VCVTPH2UDQ __m128i _mm_mask_cvtph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m128i _mm_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m256i _mm256_cvtph_epu32 (__m128h a);
VCVTPH2UDQ __m256i _mm256_mask_cvtph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m256i _mm256_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m512i _mm512_cvtph_epu32 (__m256h a);
VCVTPH2UDQ __m512i _mm512_mask_cvtph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2UDQ __m512i _mm512_maskz_cvtph_epu32 (__mmask16 k, __m256h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 79 /r VCVTPH2UQQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two unsigned quadword integers, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 79 /r VCVTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned quadword integers, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 79 /r VCVTPH2UQQ zmm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned quadword integers, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned quadword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

```
VCVTPH2UQQ DEST, SRC
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_unsigned_integer64(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPH2UQQ __m512i _mm512_cvt_roundph_epu64 (__m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_mask_cvt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_maskz_cvt_roundph_epu64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m128i _mm_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m128i _mm_mask_cvtph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m128i _mm_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m256i _mm256_mask_cvtph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m512i _mm512_mask_cvtph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 7D /r VCVTPH2UW xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m128/m16bcst to unsigned word integers, and store the result in xmm1.
EVEX.256.NP.MAP5.W0 7D /r VCVTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in ymm2/m256/m16bcst to unsigned word integers, and store the result in ymm1.
EVEX.512.NP.MAP5.W0 7D /r VCVTPH2UW zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert packed FP16 values in zmm2/m512/m16bcst to unsigned word integers, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2UW DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_unsigned_integer16(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UW __m512i _mm512_cvt_roundph_epu16 (__m512h a, int sae);
VCVTPH2UW __m512i _mm512_mask_cvt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTPH2UW __m512i _mm512_maskz_cvt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTPH2UW __m128i _mm_cvtph_epu16 (__m128h a);
VCVTPH2UW __m128i _mm_mask_cvtph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UW __m128i _mm_maskz_cvtph_epu16 (__mmask8 k, __m128h a);
VCVTPH2UW __m256i _mm256_cvtph_epu16 (__m256h a);
VCVTPH2UW __m256i _mm256_mask_cvtph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2UW __m256i _mm256_maskz_cvtph_epu16 (__mmask16 k, __m256h a);
VCVTPH2UW __m512i _mm512_cvtph_epu16 (__m512h a);
VCVTPH2UW __m512i _mm512_mask_cvtph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2UW __m512i _mm512_maskz_cvtph_epu16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7D /r VCVTPH2W xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in xmm2/m128/m16bcst to signed word integers, and store the result in xmm1.
EVEX.256.66.MAP5.W0 7D /r VCVTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert packed FP16 values in ymm2/m256/m16bcst to signed word integers, and store the result in ymm1.
EVEX.512.66.MAP5.W0 7D /r VCVTPH2W zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert packed FP16 values in zmm2/m512/m16bcst to signed word integers, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 8000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTPH2W DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_integer16(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2W __m512i _mm512_cvt_roundph_epi16 (__m512h a, int rounding);
VCVTPH2W __m512i _mm512_mask_cvt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int rounding);
VCVTPH2W __m512i _mm512_maskz_cvt_roundph_epi16 (__mmask32 k, __m512h a, int rounding);
VCVTPH2W __m128i _mm_cvtph_epi16 (__m128h a);
VCVTPH2W __m128i _mm_mask_cvtph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2W __m128i _mm_maskz_cvtph_epi16 (__mmask8 k, __m128h a);
VCVTPH2W __m256i _mm256_cvtph_epi16 (__m256h a);
VCVTPH2W __m256i _mm256_mask_cvtph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2W __m256i _mm256_maskz_cvtph_epi16 (__mmask16 k, __m256h a);
VCVTPH2W __m512i _mm512_cvtph_epi16 (__m512h a);
VCVTPH2W __m512i _mm512_mask_cvtph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2W __m512i _mm512_maskz_cvtph_epi16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTPS2PH—Convert Single Precision FP Value to 16-bit FP Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64, xmm2, imm8	A	V/V	F16C	Convert four packed single precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128, ymm2, imm8	A	V/V	F16C	Convert eight packed single precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64 {k1}{z}, xmm2, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
EVEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.512.66.0F3A.W0 1D /r ib VCVTPS2PH ymm1/m256 {k1}{z}, zmm2 {sae}, imm8	B	V/V	AVX512F OR AVX10.1	Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

Convert packed single precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e., tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

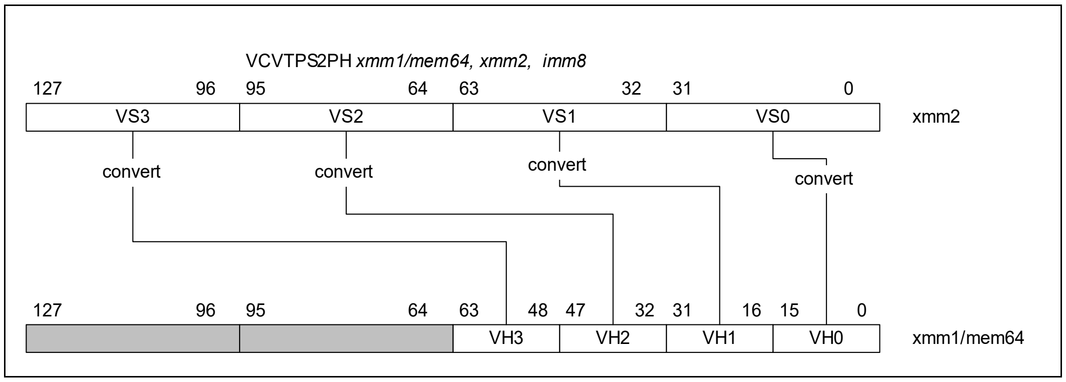


Figure 5-7. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-3.

Table 5-3. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAXVL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAXVL-1:256/128/64) of the corresponding destination register are zeroed.

Operation

```
vCvt_s2h(SRC1[31:0])
{
IF Imm[2] = 0
THEN    ; using Imm[1:0] for rounding control, see Table 5-3
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
ELSE    ; using MXCSR.RC for rounding control
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
FI;
}
```

VCVTPS2PH (EVEX Encoded Versions) When DEST is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            vCvt_s2h(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTPS2PH (EVEX Encoded Versions) When DEST is Memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            vCvt_s2h(SRC[k+31:k])
        ELSE
            *DEST[i+15:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VCVTPS2PH (VEX.256 Encoded Version)

```
DEST[15:0] := vCvt_s2h(SRC1[31:0]);
DEST[31:16] := vCvt_s2h(SRC1[63:32]);
DEST[47:32] := vCvt_s2h(SRC1[95:64]);
DEST[63:48] := vCvt_s2h(SRC1[127:96]);
DEST[79:64] := vCvt_s2h(SRC1[159:128]);
DEST[95:80] := vCvt_s2h(SRC1[191:160]);
DEST[111:96] := vCvt_s2h(SRC1[223:192]);
DEST[127:112] := vCvt_s2h(SRC1[255:224]);
DEST[MAXVL-1:128] := 0
```

VCVTPS2PH (VEX.128 Encoded Version)

```
DEST[15:0] := vCvt_s2h(SRC1[31:0]);
DEST[31:16] := vCvt_s2h(SRC1[63:32]);
DEST[47:32] := vCvt_s2h(SRC1[95:64]);
DEST[63:48] := vCvt_s2h(SRC1[127:96]);
DEST[MAXVL-1:64] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2PH __m256i _mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i _mm512_mask_cvtps_ph(__m256i s, __mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_maskz_cvtps_ph(__mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_mask_cvt_roundps_ph(__m256i s, __mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_maskz_cvt_roundps_ph(__mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m128i _mm256_mask_cvtps_ph(__m128i s, __mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm256_maskz_cvtps_ph(__mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm_mask_cvtps_ph(__m128i s, __mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_maskz_cvtps_ph(__mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_cvtps_ph ( __m128 m1, const int imm);
VCVTPS2PH __m128i _mm256_cvtps_ph(__m256 m1, const int imm);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0).

Other Exceptions

VEX-encoded instructions, see Table 2-26, "Type 11 Class Exception Conditions" (do not report #AC);

EVEX-encoded instructions, see Table 2-62, "Type E11 Class Exception Conditions."

Additionally:

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPS2PHX—Convert Packed Single Precision Floating-Point Values to Packed FP16 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 1D /r VCVTPS2PHX xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP5.W0 1D /r VCVTPS2PHX xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed single precision floating-point values in ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.66.MAP5.W0 1D /r VCVTPS2PHX ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed single precision floating-point values in zmm2 /m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed single precision floating values in the source operand to FP16 values and stores to the destination operand.

The VCVTPS2PHX instruction supports broadcasting.

This instruction uses MXCSR.DAZ for handling FP32 inputs. FP16 outputs can be normal or denormal numbers, and are not conditionally flushed based on MXCSR settings.

Operation

VCVTPS2PHX DEST, SRC (AVX512-FP16 Load Version With Broadcast Support)

```
VL = 128, 256, or 512
KL := VL / 32

IF *SRC is a register* and (VL == 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp32[0]
        ELSE
            tsrc := SRC.fp32[j]
        DEST.fp16[j] := Convert_fp32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/2] := 0
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PHX __m256h _mm512_cvtx_roundps_ph (__m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_mask_cvtx_roundps_ph (__m256h src, __mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_maskz_cvtx_roundps_ph (__mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m128h _mm_cvtxps_ph (__m128 a);
VCVTPS2PHX __m128h _mm_mask_cvtxps_ph (__m128h src, __mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm_maskz_cvtxps_ph (__mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm256_cvtxps_ph (__m256 a);
VCVTPS2PHX __m128h _mm256_mask_cvtxps_ph (__m128h src, __mmask8 k, __m256 a);
VCVTPS2PHX __m128h _mm256_maskz_cvtxps_ph (__mmask8 k, __m256 a);
VCVTPS2PHX __m256h _mm512_cvtxps_ph (__m512 a);
VCVTPS2PHX __m256h _mm512_mask_cvtxps_ph (__m256h src, __mmask16 k, __m512 a);
VCVTPS2PHX __m256h _mm512_maskz_cvtxps_ph (__mmask16 k, __m512 a);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0).

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts eight packed single precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 64
   k := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] :=
         Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
      ELSE
         IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
               DEST[i+63:i] := 0
         FI
```

```
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2QQ __m512i _mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvt_roundps_epi64( __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i _mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i _mm_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm_maskz_cvtps_epi64( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPS2UDQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.
EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.
EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512F OR AVX10.1	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts sixteen packed single precision floating-point values in the source operand to sixteen unsigned double-word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no *
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2UDQ __m512i _mm512_cvtps_epu32( __m512 a);
VCVTPS2UDQ __m512i _mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_cvt_roundps_epu32( __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i _mm256_cvtps_epu32( __m256d a);
VCVTPS2UDQ __m256i _mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i _mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i _mm_cvtps_epu32( __m128 a);
VCVTPS2UDQ __m128i _mm_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i _mm_maskz_cvtps_epu32( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.
EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts up to eight packed single precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2UQQ __m512i _mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i _mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i _mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i _mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i _mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i _mm_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i _mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed quadword integers from xmm2/m128/m64bcst to packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed quadword integers from ymm2/m256/m64bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed quadword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PD (EVEX2 Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTQQ2PD (EVEX Encoded Versions) when SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTQQ2PD __m512d _mm512_cvtepi64_pd( __m512i a);
VCVTQQ2PD __m512d _mm512_mask_cvtepi64_pd( __m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_maskz_cvtepi64_pd( __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_cvt_roundepi64_pd( __m512i a, int r);
VCVTQQ2PD __m512d _mm512_mask_cvt_roundepi64_pd( __m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m512d _mm512_maskz_cvt_roundepi64_pd( __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m256d _mm256_mask_cvtepi64_pd( __m256d s, __mmask8 k, __m256i a);
VCVTQQ2PD __m256d _mm256_maskz_cvtepi64_pd( __mmask8 k, __m256i a);
VCVTQQ2PD __m128d _mm_mask_cvtepi64_pd( __m128d s, __mmask8 k, __m128i a);
VCVTQQ2PD __m128d _mm_maskz_cvtepi64_pd( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTQQ2PH—Convert Packed Signed Quadword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert two packed signed quadword integers in xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed signed quadword integers in ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert eight packed signed quadword integers in zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed signed quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTQQ2PH DEST, SRC

```
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.qword[0]
        ELSE
            tsrc := SRC.qword[j]
        DEST.fp16[j] := Convert_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PH __m128h _mm512_cvt_roundepi64_ph (__m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_mask_cvt_roundepi64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_maskz_cvt_roundepi64_ph (__mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm_cvtepi64_ph (__m128i a);
VCVTQQ2PH __m128h _mm_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm_maskz_cvtepi64_ph (__mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm256_cvtepi64_ph (__m256i a);
VCVTQQ2PH __m128h _mm256_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm256_maskz_cvtepi64_ph (__mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm512_cvtepi64_ph (__m512i a);
VCVTQQ2PH __m128h _mm512_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTQQ2PH __m128h _mm512_maskz_cvtepi64_ph (__mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed quadword integers from xmm2/mem to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed quadword integers from ymm2/mem to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.512.0F.W1 5B /r VCVTQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed quadword integers from zmm2/mem to eight packed single precision floating-point values in ymm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed quadword integers in the source operand (second operand) to packed single precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[k+31:k] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[k+31:k] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[k+31:k] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[k+31:k] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTQQ2PS __m256 _mm512_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 _mm512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQQ2PS __m256 _mm512_mask_cvt_roundepi_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 _mm512_maskz_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 _mm256_cvtepi64_ps( __m256i a);
VCVTQQ2PS __m128 _mm256_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm256_maskz_cvtepi64_ps( __mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm_cvtepi64_ps( __m128i a);
VCVTQQ2PS __m128 _mm_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 _mm_maskz_cvtepi64_ps( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTSD2SH—Convert Low FP64 Value to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP5.W1 5A /r VCVTSD2SH xmm1{k1}{z}, xmm2, xmm3/m64 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert the low FP64 value in xmm3/m64 to an FP16 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP64 value in the second source operand to an FP16 value, and stores the result in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VCVTSD2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := Convert_fp64_to_fp16(SRC2.fp64[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2SH __m128h _mm_cvt_roundsd_sh (__m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_mask_cvt_roundsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_maskz_cvt_roundsd_sh (__mmask8 k, __m128h a, __m128d b, const int rounding);
VCVTSD2SH __m128h _mm_cvtsd_sh (__m128h a, __m128d b);
VCVTSD2SH __m128h _mm_mask_cvtsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b);
VCVTSD2SH __m128h _mm_maskz_cvtsd_sh (__mmask8 k, __m128h a, __m128d b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCVTSD2USI—Convert Scalar Double Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}	A	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.
EVEX.LLIG.F2.0F.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.

NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

Operation

VCVTSD2USI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvt_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvt_roundsd_u64(__m128d, int r);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTSH2SD—Convert Low FP16 Value to an FP64 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5A /r VCVTSH2SD xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert the low FP16 value in xmm3/m16 to an FP64 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:64 of xmm2 are copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP16 element in the second source operand to a FP64 element in the low element of the destination operand.

Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP64 element of the destination is updated according to the writemask.

Operation

VCVTSH2SD dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp64[0] := Convert_fp16_to_fp64(SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp64[0] := 0
// else dest.fp64[0] remains unchanged

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSH2SD __m128d _mm_cvt_roundsh_sd (__m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_mask_cvt_roundsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_maskz_cvt_roundsh_sd (__mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d _mm_cvtsh_sd (__m128d a, __m128h b);
VCVTSH2SD __m128d _mm_mask_cvtsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b);
VCVTSH2SD __m128d _mm_maskz_cvtsh_sd (__mmask8 k, __m128d a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCVTSH2SI—Convert Low FP16 Value to Signed Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2D /r VCVTSH2SI r32, xmm1/m16 {er}	A	V/V[1]	AVX512-FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to a signed doubleword integer and store the result in r32.
EVEX.LLIG.F3.MAP5.W1 2D /r VCVTSH2SI r64, xmm1/m16 {er}	A	V/N.E.	AVX512-FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to a signed quadword integer and store the result in r64.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Operation

```
VCVTSH2SI dest, src
IF *SRC is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_integer64(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_integer32(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSH2SI int _mm_cvt_roundsh_i32 (__m128h a, int rounding);
VCVTSH2SI __int64 _mm_cvt_roundsh_i64 (__m128h a, int rounding);
VCVTSH2SI int _mm_cvtsh_i32 (__m128h a);
VCVTSH2SI __int64 _mm_cvtsh_i64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTSH2SS—Convert Low FP16 Value to FP32 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP6.W0 13 /r VCVTSH2SS xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert the low FP16 element in xmm3/m16 to an FP32 value and store in the low element of xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP16 element in the second source operand to the low FP32 element of the destination operand.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VCVTSH2SS dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp32[0] := Convert_fp16_to_fp32(SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp32[0] := 0
// else dest.fp32[0] remains unchanged

DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSH2SS __m128 _mm_cvt_roundsh_ss (__m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_mask_cvt_roundsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_maskz_cvt_roundsh_ss (__mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 _mm_cvtsh_ss (__m128 a, __m128h b);
VCVTSH2SS __m128 _mm_mask_cvtsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b);
VCVTSH2SS __m128 _mm_maskz_cvtsh_ss (__mmask8 k, __m128 a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 79 /r VCVTSH2USI r32, xmm1/m16 {er}	A	V/V[1]	AVX512-FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to an unsigned doubleword integer and store the result in r32.
EVEX.LLIG.F3.MAP5.W1 79 /r VCVTSH2USI r64, xmm1/m16 {er}	A	V/N.E.	AVX512-FP16 OR AVX10.1	Convert the low FP16 element in xmm1/m16 to an unsigned quadword integer and store the result in r64.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

Operation

VCVTSH2USI dest, src

```
// SET_RM() sets the rounding mode used for this instruction.
IF *SRC is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_unsigned_integer64(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_unsigned_integer32(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSH2USI unsigned int _mm_cvt_roundsh_u32 (__m128h a, int sae);
VCVTSH2USI unsigned __int64 _mm_cvt_roundsh_u64 (__m128h a, int rounding);
VCVTSH2USI unsigned int _mm_cvtsh_u32 (__m128h a);
VCVTSH2USI unsigned __int64 _mm_cvtsh_u64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTSI2SH—Convert a Signed Doubleword/Quadword Integer to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2A /r VCVTSI2SH xmm1, xmm2, r32/m32 {er}	A	V/V[1]	AVX512-FP16 OR AVX10.1	Convert the signed doubleword integer in r32/m32 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].
EVEX.LLIG.F3.MAP5.W1 2A /r VCVTSI2SH xmm1, xmm2, r64/m64 {er}	A	V/N.E.	AVX512-FP16 OR AVX10.1	Convert the signed quadword integer in r64/m64 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to an FP16 value in the destination operand. The result is stored in the low word of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand. Bits MAXVL-1:128 of the destination register are zeroed.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTSI2SH dest, src1, src2

```
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
    DEST.fp16[0] := Convert_integer64_to_fp16(SRC2.qword)
ELSE:
    DEST.fp16[0] := Convert_integer32_to_fp16(SRC2.dword)

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SH __m128h _mm_cvt_roundi32_sh (__m128h a, int b, int rounding);
VCVTSI2SH __m128h _mm_cvt_roundi64_sh (__m128h a, __int64 b, int rounding);
VCVTSI2SH __m128h _mm_cvti32_sh (__m128h a, int b);
VCVTSI2SH __m128h _mm_cvti64_sh (__m128h a, __int64 b);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

VCVTSS2SH—Convert Low FP32 Value to an FP16 Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 1D /r VCVTSS2SH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert low FP32 value in xmm3/m32 to an FP16 value and store in the low element of xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction converts the low FP32 value in the second source operand to a FP16 value in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
VCVTSS2SH dest, src1, src2
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := Convert_fp32_to_fp16(SRC2.fp32[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSS2SH __m128h _mm_cvt_roundss_sh (__m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_mask_cvt_roundss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_maskz_cvt_roundss_sh (__mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h _mm_cvtss_sh (__m128h a, __m128 b);
VCVTSS2SH __m128h _mm_mask_cvtss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b);
VCVTSS2SH __m128h _mm_maskz_cvtss_sh (__mmask8 k, __m128h a, __m128 b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VCVTSS2USI—Convert Scalar Single Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}	A	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.
EVEX.LLIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.

NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTSS2USI (EVEX Encoded Version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned _mm_cvtss_u32(__m128 a);
VCVTSS2USI unsigned _mm_cvt_roundss_u32(__m128 a, int r);
VCVTSS2USI unsigned __int64 _mm_cvtss_u64(__m128 a);
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTTPD2QQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from zmm2/m128/m64bcst to two packed signed quadword integers in zmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 7A /r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed signed quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 7A /r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/m512 to eight packed signed quadword integers in zmm1 using truncation with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed double precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=         Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPD2QQ __m512i _mm512_cvttpd_epi64( __m512d a);
VCVTTPD2QQ __m512i _mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i _mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i _mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i _mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i _mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPD2UDQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 78 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W1 78 02 /r VCVTTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.0F.W1 78 /r VCVTTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512F OR AVX10.1	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed double precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256),(8, 512)

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPD2UDQ __m256i _mm512_cvttpd_epu32( __m512d a);
VCVTTPD2UDQ __m256i _mm512_mask_cvttpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_maskz_cvttpd_epu32( __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_cvtt_roundpd_epu32( __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_mask_cvtt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_maskz_cvtt_roundpd_epu32( __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i _mm256_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvttpd_epu32( __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i _mm_maskz_cvttpd_epu32( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPD2UQQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 78 /r VCVTTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed double precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 78 /r VCVTTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 78 /r VCVTTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed double precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
         THEN
              IF (EVEX.b == 1)
                   THEN
                        DEST[i+63:i] :=
              Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
                   ELSE
                        DEST[i+63:i] :=
              Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
              FI;
         ELSE
              IF *merging-masking*                    ; merging-masking
                   THEN *DEST[i+63:i] remains unchanged*
                   ELSE                               ; zeroing-masking
                        DEST[i+63:i] := 0
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPD2UQQ _mm<size>[_mask[z]]_cvtt[_round]pd_epu64
VCVTTPD2UQQ __m512i _mm512_cvttpd_epu64( __m512d a);
VCVTTPD2UQQ __m512i _mm512_mask_cvttpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_maskz_cvttpd_epu64( __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_cvtt_roundpd_epu64( __m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_mask_cvtt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_maskz_cvtt_roundpd_epu64( __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i _mm256_mask_cvttpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i _mm256_maskz_cvttpd_epu64( __mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i _mm_mask_cvttpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i _mm_maskz_cvttpd_epu64( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPH2DQ—Convert with Truncation Packed FP16 Values to Signed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.MAP5.W0 5B /r VCVTTPH2DQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.MAP5.W0 5B /r VCVTTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.MAP5.W0 5B /r VCVTTPH2DQ zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed doubleword integers in destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2DQ dest, src

```
VL = 128, 256 or 512
KL := VL / 32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp32[j] := Convert_fp16_to_integer32_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.fp32[j] := 0
    // else dest.fp32[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2DQ __m512i _mm512_cvtt_roundph_epi32 (__m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_mask_cvtt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_maskz_cvtt_roundph_epi32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m128i _mm_cvttph_epi32 (__m128h a);
VCVTTPH2DQ __m128i _mm_mask_cvttph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m128i _mm_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_cvttph_epi32 (__m128h a);
VCVTTPH2DQ __m256i _mm256_mask_cvttph_epi32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m512i _mm512_cvttph_epi32 (__m256h a);
VCVTTPH2DQ __m512i _mm512_mask_cvttph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2DQ __m512i _mm512_maskz_cvttph_epi32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2QQ—Convert with Truncation Packed FP16 Values to Signed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7A /r VCVTTPH2QQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 7A /r VCVTTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 7A /r VCVTTPH2QQ zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2QQ dest, src

```
VL = 128, 256 or 512
KL := VL / 64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_integer64_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2QQ __m512i _mm512_cvtt_roundph_epi64 (__m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_mask_cvtt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_maskz_cvtt_roundph_epi64 (__mmask8 k, __m128h a, int sae);
VCVTTPH2QQ __m128i _mm_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m128i _mm_mask_cvttph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m128i _mm_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTPH2QQ __m256i _mm256_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m256i _mm256_mask_cvttph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m256i _mm256_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTPH2QQ __m512i _mm512_cvttph_epi64 (__m128h a);
VCVTTPH2QQ __m512i _mm512_mask_cvttph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTTPH2QQ __m512i _mm512_maskz_cvttph_epi64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2UDQ—Convert with Truncation Packed FP16 Values to Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 78 /r VCVTTPH2UDQ xmm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.NP.MAP5.W0 78 /r VCVTTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.NP.MAP5.W0 78 /r VCVTTPH2UDQ zmm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UDQ dest, src

```
VL = 128, 256 or 512
KL := VL / 32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.dword[j] := Convert_fp16_to_unsigned_integer32_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPH2UDQ __m512i _mm512_cvtt_roundph_epu32 (__m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_mask_cvtt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_maskz_cvtt_roundph_epu32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m128i _mm_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m128i _mm_mask_cvttph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m128i _mm_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m256i _mm256_mask_cvttph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m512i _mm512_cvttph_epu32 (__m256h a);
VCVTTPH2UDQ __m512i _mm512_mask_cvttph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2UDQ __m512i _mm512_maskz_cvttph_epu32 (__mmask16 k, __m256h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2UQQ—Convert with Truncation Packed FP16 Values to Unsigned Quadword Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 78 /r VCVTTPH2UQQ xmm1{k1}{z}, xmm2/m32/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert two packed FP16 values in xmm2/m32/m16bcst to two unsigned quadword integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 78 /r VCVTTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned quadword integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 78 /r VCVTTPH2UQQ zmm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned quadword integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UQQ dest, src

```
VL = 128, 256 or 512
KL := VL / 64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_unsigned_integer64_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPH2UQQ __m512i _mm512_cvtt_roundph_epu64 (__m128h a, int sae);
VCVTTPH2UQQ __m512i _mm512_mask_cvtt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTTPH2UQQ __m512i _mm512_maskz_cvtt_roundph_epu64 (__mmask8 k, __m128h a, int sae);
VCVTTPH2UQQ __m128i _mm_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m128i _mm_mask_cvttph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m128i _mm_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTTPH2UQQ __m256i _mm256_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m256i _mm256_mask_cvttph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m256i _mm256_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTTPH2UQQ __m512i _mm512_cvttph_epu64 (__m128h a);
VCVTTPH2UQQ __m512i _mm512_mask_cvttph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTTPH2UQQ __m512i _mm512_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 7C /r VCVTTPH2UW xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned word integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.NP.MAP5.W0 7C /r VCVTTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned word integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.NP.MAP5.W0 7C /r VCVTTPH2UW zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert thirty-two packed FP16 values in zmm2/m512/m16bcst to thirty-two unsigned word integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFH is returned.

The destination elements are updated according to the writemask.

Operation

VCVTTPH2UW dest, src

```
VL = 128, 256 or 512
KL := VL / 16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_unsigned_integer16_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPH2UW __m512i _mm512_cvtt_roundph_epu16 (__m512h a, int sae);
VCVTTPH2UW __m512i _mm512_mask_cvtt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m512i _mm512_maskz_cvtt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m128i _mm_cvttph_epu16 (__m128h a);
VCVTTPH2UW __m128i _mm_mask_cvttph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UW __m128i _mm_maskz_cvttph_epu16 (__mmask8 k, __m128h a);
VCVTTPH2UW __m256i _mm256_cvttph_epu16 (__m256h a);
VCVTTPH2UW __m256i _mm256_mask_cvttph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2UW __m256i _mm256_maskz_cvttph_epu16 (__mmask16 k, __m256h a);
VCVTTPH2UW __m512i _mm512_cvttph_epu16 (__m512h a);
VCVTTPH2UW __m512i _mm512_mask_cvttph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2UW __m512i _mm512_maskz_cvttph_epu16 (__mmask32 k, __m512h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.W0 7C /r VCVTTPH2W xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed word integers, and store the result in xmm1 using truncation subject to writemask k1.
EVEX.256.66.MAP5.W0 7C /r VCVTTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen signed word integers, and store the result in ymm1 using truncation subject to writemask k1.
EVEX.512.66.MAP5.W0 7C /r VCVTTPH2W zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert thirty-two packed FP16 values in zmm2/m512/m16bcst to thirty-two signed word integers, and store the result in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed FP16 values in the source operand to signed word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value 8000H is returned.

The destination elements are updated according to the writemask.

Operation

```
VCVTTPH2W dest, src
VL = 128, 256 or 512
KL := VL / 16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_integer16_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2W __m512i _mm512_cvtt_roundph_epi16 (__m512h a, int sae);
VCVTTPH2W __m512i _mm512_mask_cvtt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2W __m512i _mm512_maskz_cvtt_roundph_epi16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2W __m128i _mm_cvttph_epi16 (__m128h a);
VCVTTPH2W __m128i _mm_mask_cvttph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2W __m128i _mm_maskz_cvttph_epi16 (__mmask8 k, __m128h a);
VCVTTPH2W __m256i _mm256_cvttph_epi16 (__m256h a);
VCVTTPH2W __m256i _mm256_mask_cvttph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2W __m256i _mm256_maskz_cvttph_epi16 (__mmask16 k, __m256h a);
VCVTTPH2W __m512i _mm512_cvttph_epi16 (__m512h a);
VCVTTPH2W __m512i _mm512_mask_cvttph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2W __m512i _mm512_maskz_cvttph_epi16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTTPS2QQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7A /r VCVTTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 7A /r VCVTTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 7A /r VCVTTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed single precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
         THEN
              IF (EVEX.b == 1)
                   THEN
                        DEST[i+63:i] :=
              Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
                   ELSE
                        DEST[i+63:i] :=
              Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
              FI;
         ELSE
              IF *merging-masking*                    ; merging-masking
                   THEN *DEST[i+63:i] remains unchanged*
                   ELSE                               ; zeroing-masking
                        DEST[i+63:i] := 0
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPS2QQ __m512i _mm512_cvttps_epi64( __m256 a);
VCVTTPS2QQ __m512i _mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_maskz_cvtt_roundps_epi64( __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i _mm256_mask_cvttps_epi64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i _mm256_maskz_cvttps_epi64( __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_mask_cvttps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_maskz_cvttps_epi64( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPS2UDQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512F OR AVX10.1	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation packed single precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] :=
         Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
      ELSE
         IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                 ; zeroing-masking
               DEST[i+31:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPS2UDQ __m512i _mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i _mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i _mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i _mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i _mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTPS2UQQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 78 /r VCVTTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 78 /r VCVTTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 78 /r VCVTTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation up to eight packed single precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTPS2UQQ _mm<size>[_mask[z]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ __m512i _mm512_cvttps_epu64( __m256 a);
VCVTTPS2UQQ __m512i _mm512_mask_cvttps_epu64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i _mm512_maskz_cvttps_epu64( __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i _mm512_cvtt_roundps_epu64( __m256 a, int sae);
VCVTTPS2UQQ __m512i _mm512_mask_cvtt_roundps_epu64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i _mm512_maskz_cvtt_roundps_epu64( __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i _mm256_mask_cvttps_epu64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m256i _mm256_maskz_cvttps_epu64( __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i _mm_mask_cvttps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i _mm_maskz_cvttps_epu64( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTTSD2USI—Convert With Truncation Scalar Double Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}	A	V/V	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.
EVEX.LLIG.F2.0F.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation a double precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Operation

VCVTTSD2USI (EVEX Encoded Version)

```
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSD2USI unsigned int _mm_cvttsd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttsd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundsd_u64(__m128d, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTTSH2SI—Convert with Truncation Low FP16 Value to a Signed Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 2C /r VCVTTSH2SI r32, xmm1/m16 {sae}	A	V/V[1]	AVX512-FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to a signed doubleword integer and store the result in r32 using truncation.
EVEX.LLIG.F3.MAP5.W1 2C /r VCVTTSH2SI r64, xmm1/m16 {sae}	A	V/N.E.	AVX512-FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to a signed quadword integer and store the result in r64 using truncation.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000H is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value 80000000_00000000H is returned.

Operation

```
VCVTTSH2SI dest, src
IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_integer64_truncate(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_integer32_truncate(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSH2SI int _mm_cvtt_roundsh_i32 (__m128h a, int sae);
VCVTTSH2SI __int64 _mm_cvtt_roundsh_i64 (__m128h a, int sae);
VCVTTSH2SI int _mm_cvttsh_i32 (__m128h a);
VCVTTSH2SI __int64 _mm_cvttsh_i64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTTSH2USI—Convert with Truncation Low FP16 Value to an Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 78 /r VCVTTSH2USI r32, xmm1/m16 {sae}	A	V/V[1]	AVX512-FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to an unsigned doubleword integer and store the result in r32 using truncation.
EVEX.LLIG.F3.MAP5.W1 78 /r VCVTTSH2USI r64, xmm1/m16 {sae}	A	V/N.E.	AVX512-FP16 OR AVX10.1	Convert FP16 value in the low element of xmm1/m16 to an unsigned quadword integer and store the result in r64 using truncation.

NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

Operation

VCVTTSH2USI dest, src

```
IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_unsigned_integer64_truncate(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_unsigned_integer32_truncate(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSH2USI unsigned int _mm_cvtt_roundsh_u32 (__m128h a, int sae);
VCVTTSH2USI unsigned __int64 _mm_cvtt_roundsh_u64 (__m128h a, int sae);
VCVTTSH2USI unsigned int _mm_cvttsh_u32 (__m128h a);
VCVTTSH2USI unsigned __int64 _mm_cvttsh_u64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTTSS2USI—Convert With Truncation Scalar Single Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}	A	V/V	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.
EVEX.LLIG.F3.0F.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts with truncation a single precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) value is returned.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFFH is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the integer value FFFFFFFF_FFFFFFFFH is returned.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTSS2USI (EVEX Encoded Version)

```
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSS2USI unsigned int _mm_cvttss_u32( __m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32( __m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64( __m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64( __m128 a, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 7A /r VCVTUDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double precision floating-point values in zmm1 with writemask k1.
EVEX.256.F3.0F.W0 7A /r VCVTUDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double precision floating-point values in zmm1 with writemask k1.
EVEX.512.F3.0F.W0 7A /r VCVTUDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512F OR AVX10.1	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUDQ2PD __m512d _mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d _mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d _mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d _mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d _mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d _mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d _mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-53, “Type E5 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.F2.MAP5.W0 7A /r VCVTUDQ2PH ymm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned doubleword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUDQ2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_unsigned_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUDQ2PH __m256h _mm512_cvt_roundepu32_ph (__m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_mask_cvt_roundepu32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_maskz_cvt_roundepu32_ph (__mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m128h _mm_cvtepu32_ph (__m128i a);
VCVTUDQ2PH __m128h _mm_mask_cvtepu32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUDQ2PH __m128h _mm_maskz_cvtepu32_ph (__mmask8 k, __m128i a);
VCVTUDQ2PH __m128h _mm256_cvtepu32_ph (__m256i a);
VCVTUDQ2PH __m128h _mm256_mask_cvtepu32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTUDQ2PH __m128h _mm256_maskz_cvtepu32_ph (__mmask8 k, __m256i a);
VCVTUDQ2PH __m256h _mm512_cvtepu32_ph (__m512i a);
VCVTUDQ2PH __m256h _mm512_mask_cvtepu32_ph (__m256h src, __mmask16 k, __m512i a);
VCVTUDQ2PH __m256h _mm512_maskz_cvtepu32_ph (__mmask16 k, __m512i a);
```

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single precision floating-point values in zmm1 with writemask k1.
EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX512F OR AVX10.1	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned doubleword integers in the source operand (second operand) to single precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUDQ2PS __m512 _mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 _mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_mask_cvt_roundepu32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_maskz_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 _mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 _mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 _mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 _mm_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 _mm_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 _mm_maskz_cvtepu32_ps( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 7A /r VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 7A /r VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 7A /r VCVTUQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUQQ2PD __m512d _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d _mm_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUQQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert two packed unsigned doubleword integers from xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert four packed unsigned doubleword integers from ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.512.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert eight packed unsigned doubleword integers from zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

```
VCVTUQQ2PH dest, src
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.qword[0]
        ELSE
            tsrc := SRC.qword[j]
        DEST.fp16[j] := Convert_unsigned_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PH __m128h _mm512_cvt_roundepu64_ph (__m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_mask_cvt_roundepu64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_maskz_cvt_roundepu64_ph (__mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm_cvtepu64_ph (__m128i a);
VCVTUQQ2PH __m128h _mm_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm_maskz_cvtepu64_ph (__mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm256_cvtepu64_ph (__m256i a);
VCVTUQQ2PH __m128h _mm256_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm256_maskz_cvtepu64_ph (__mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm512_cvtepu64_ph (__m512i a);
VCVTUQQ2PH __m128h _mm512_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTUQQ2PH __m128h _mm512_maskz_cvtepu64_ph (__mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single precision floating-point values in zmm1 with writemask k1.
EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.512.F2.0F.W1 7A /r VCVTUQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst {er}	A	V/V	AVX512DQ OR AVX10.1	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed unsigned quadword integers in the source operand (second operand) to single precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+31:i] :=
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUQQ2PS __m256 _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PS __m256 _mm512_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_mask_cvt_roundepu64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m128 _mm256_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PS __m128 _mm_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 _mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VCVTUSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F.W0 7B /r VCVTUSI2SD xmm1, xmm2, r/m32	A	V/V	AVX512F OR AVX10.1	Convert one unsigned doubleword integer from r/m32 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W1 7B /r VCVTUSI2SD xmm1, xmm2, r/m64{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one unsigned quadword integer from r/m64 to one double precision floating-point value in xmm1.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

Operation

VCVTUSI2SD (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SD __m128d _mm_cvtu32_sd(__m128d s, unsigned a);
VCVTUSI2SD __m128d _mm_cvtu64_sd(__m128d s, unsigned __int64 a);
VCVTUSI2SD __m128d _mm_cvt_roundu64_sd(__m128d s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 2-50, “Type E3NF Class Exception Conditions” if W1; otherwise, see Table 2-61, “Type E10NF Class Exception Conditions.”

VCVTUSI2SS—Convert Unsigned Integer to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}	A	V/V	AVX512F OR AVX10.1	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}	A	V/N.E.[1]	AVX512F OR AVX10.1	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

Operation

VCVTUSI2SS (EVEX Encoded Version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS __m128 _mm_cvtu32_ss(__m128 s, unsigned a);
VCVTUSI2SS __m128 _mm_cvt_roundu32_ss(__m128 s, unsigned a, int r);
VCVTUSI2SS __m128 _mm_cvtu64_ss(__m128 s, unsigned __int64 a);
VCVTUSI2SS __m128 _mm_cvt_roundu64_ss(__m128 s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 2-50, “Type E3NF Class Exception Conditions.”

VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP5.W0 7D /r VCVTUW2PH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed unsigned word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP5.W0 7D /r VCVTUW2PH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed unsigned word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP5.W0 7D /r VCVTUW2PH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert thirty-two packed unsigned word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed unsigned word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

VCVTUW2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_unsignd_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUW2PH __m512h _mm512_cvt_roundepu16_ph (__m512i a, int rounding);
VCVTUW2PH __m512h _mm512_mask_cvt_roundepu16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m512h _mm512_maskz_cvt_roundepu16_ph (__mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m128h _mm_cvtepu16_ph (__m128i a);
VCVTUW2PH __m128h _mm_mask_cvtepu16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUW2PH __m128h _mm_maskz_cvtepu16_ph (__mmask8 k, __m128i a);
VCVTUW2PH __m256h _mm256_cvtepu16_ph (__m256i a);
VCVTUW2PH __m256h _mm256_mask_cvtepu16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTUW2PH __m256h _mm256_maskz_cvtepu16_ph (__mmask16 k, __m256i a);
VCVTUW2PH __m512h _mm512_cvtepu16_ph (__m512i a);
VCVTUW2PH __m512h _mm512_mask_cvtepu16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTUW2PH __m512h _mm512_maskz_cvtepu16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions

Overflow, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.MAP5.W0 7D /r VCVTW2PH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert eight packed signed word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP5.W0 7D /r VCVTW2PH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert sixteen packed signed word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP5.W0 7D /r VCVTW2PH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Convert thirty-two packed signed word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction converts packed signed word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

Operation

VCVTW2PH dest, src

```
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTW2PH __m512h _mm512_cvt_roundepi16_ph (__m512i a, int rounding);
VCVTW2PH __m512h _mm512_mask_cvt_roundepi16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTW2PH __m512h _mm512_maskz_cvt_roundepi16_ph (__mmask32 k, __m512i a, int rounding);
VCVTW2PH __m128h _mm_cvtepi16_ph (__m128i a);
VCVTW2PH __m128h _mm_mask_cvtepi16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTW2PH __m128h _mm_maskz_cvtepi16_ph (__mmask8 k, __m128i a);
VCVTW2PH __m256h _mm256_cvtepi16_ph (__m256i a);
VCVTW2PH __m256h _mm256_mask_cvtepi16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTW2PH __m256h _mm256_maskz_cvtepi16_ph (__mmask16 k, __m256i a);
VCVTW2PH __m512h _mm512_cvtepi16_ph (__m512i a);
VCVTW2PH __m512h _mm512_mask_cvtepi16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTW2PH __m512h _mm512_maskz_cvtepi16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.
EVEX.256.66.0F3A.W0 42 /r ib VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.
EVEX.512.66.0F3A.W0 42 /r ib VDBPSADBW zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see "Tmp1" in Figure 5-8, is constructed from the second source operand the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.
- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corre-sponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.
- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the inter-mediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

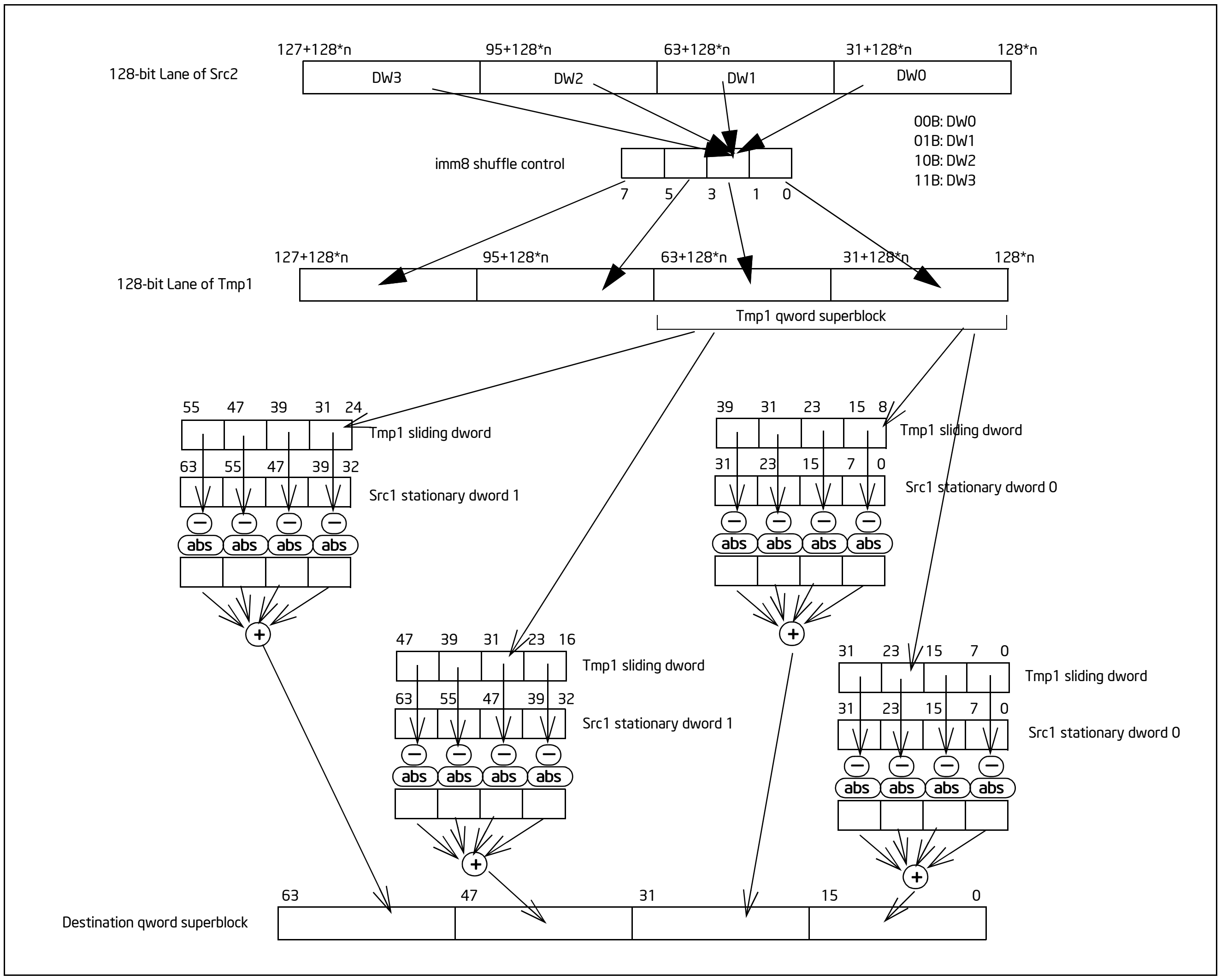


Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW

Operation

VDBPSADBW (EVEX Encoded Versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Selection of quadruplets:
FOR I = 0 to VL step 128
    TMP1[I+31:I] := select (SRC2[I+127: I], imm8[1:0])
    TMP1[I+63: I+32] := select (SRC2[I+127: I], imm8[3:2])
    TMP1[I+95: I+64] := select (SRC2[I+127: I], imm8[5:4])
    TMP1[I+127: I+96]  := select (SRC2[I+127: I], imm8[7:6])
END FOR

SAD of quadruplets:

FOR I =0 to VL step 64
    TMP_DEST[I+15:I] := ABS(SRC1[I+7: I] - TMP1[I+7: I]) +
        ABS(SRC1[I+15: I+8]- TMP1[I+15: I+8]) +
        ABS(SRC1[I+23: I+16]- TMP1[I+23: I+16]) +
        ABS(SRC1[I+31: I+24]- TMP1[I+31: I+24])

    TMP_DEST[I+31: I+16] := ABS(SRC1[I+7: I] - TMP1[I+15: I+8]) +
        ABS(SRC1[I+15: I+8]- TMP1[I+23: I+16]) +
        ABS(SRC1[I+23: I+16]- TMP1[I+31: I+24]) +
        ABS(SRC1[I+31: I+24]- TMP1[I+39: I+32])
    TMP_DEST[I+47: I+32] := ABS(SRC1[I+39: I+32] - TMP1[I+23: I+16]) +
        ABS(SRC1[I+47: I+40]- TMP1[I+31: I+24]) +
        ABS(SRC1[I+55: I+48]- TMP1[I+39: I+32]) +
        ABS(SRC1[I+63: I+56]- TMP1[I+47: I+40])

    TMP_DEST[I+63: I+48] := ABS(SRC1[I+39: I+32] - TMP1[I+31: I+24]) +
        ABS(SRC1[I+47: I+40] - TMP1[I+39: I+32]) +
        ABS(SRC1[I+55: I+48] - TMP1[I+47: I+40]) +
        ABS(SRC1[I+63: I+56] - TMP1[I+55: I+48])
ENDFOR

FOR j :=  0 TO KL-1
    i :=    j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=  TMP_DEST[i+15:i]
        ELSE
                IF *merging-masking*                    ; merging-masking
                        THEN *DEST[i+15:i] remains unchanged*
                        ELSE                            ; zeroing-masking
                                DEST[i+15:i] :=  0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] :=  0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b int imm8);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VDIVPH—Divide Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5E /r VDIVPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Divide packed FP16 values in xmm2 by packed FP16 values in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5E /r VDIVPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Divide packed FP16 values in ymm2 by packed FP16 values in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5E /r VDIVPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Divide packed FP16 values in zmm2 by packed FP16 values in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction divides packed FP16 values from the first source operand by the corresponding elements in the second source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVPH __m128h _mm_div_ph (__m128h a, __m128h b);
VDIVPH __m128h _mm_mask_div_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVPH __m128h _mm_maskz_div_ph (__mmask8 k, __m128h a, __m128h b);
VDIVPH __m256h _mm256_div_ph (__m256h a, __m256h b);
VDIVPH __m256h _mm256_mask_div_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VDIVPH __m256h _mm256_maskz_div_ph (__mmask16 k, __m256h a, __m256h b);
VDIVPH __m512h _mm512_div_ph (__m512h a, __m512h b);
VDIVPH __m512h _mm512_mask_div_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_maskz_div_ph (__mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_div_round_ph (__m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_mask_div_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_maskz_div_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VDIVSH—Divide Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5E /r VDIVSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Divide low FP16 value in xmm2 by low FP16 value in xmm3/m16, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction divides the low FP16 value from the first source operand by the corresponding value in the second source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VDIVSH (EVEX Encoded Versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] / SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDIVSH __m128h _mm_div_round_sh (__m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_mask_div_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_maskz_div_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h _mm_div_sh (__m128h a, __m128h b);
VDIVSH __m128h _mm_mask_div_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVSH __m128h _mm_maskz_div_sh (__mmask8 k, __m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 52 /r VDPBF16PS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-BF16 AND AVX512VL) OR AVX10.1	Multiply BF16 pairs from xmm2 and xmm3/m128, and accumulate the resulting packed single precision results in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 52 /r VDPBF16PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-BF16 AND AVX512VL) OR AVX10.1	Multiply BF16 pairs from ymm2 and ymm3/m256, and accumulate the resulting packed single precision results in ymm1 with writemask k1.
EVEX.512.F3.0F38.W0 52 /r VDPBF16PS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	(AVX512-BF16 AND AVX512F) OR AVX10.1	Multiply BF16 pairs from zmm2 and zmm3/m512, and accumulate the resulting packed single precision results in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD dot-product of two BF16 pairs and accumulates into a packed single precision register.

“Round to nearest even” rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

NaN propagation priorities are described in Table 5-4.

Table 5-4. NaN Propagation Priorities

NaN Priority	Description	Comments
1	src1 low is NaN	Lower part has priority over upper part, i.e., it overrides the upper part.
2	src2 low is NaN	
3	src1 high is NaN	Upper part may be overridden if lower has NaN.
4	src2 high is NaN	
5	srcdest is NaN	Dest is propagated if no NaN is encountered by src2.

Operation

```
Define make_fp32(x):
    // The x parameter is bfloat16. Pack it in to upper 16b of a dword. The bit pattern is a legal fp32 value. Return that bit pattern.
    dword := 0
    dword[31:16] := x
    RETURN dword
```

```
VDPBF16PS srcdest, src1, src2
VL = (128, 256, 512)
KL = VL/32

origdest := srcdest
FOR i := 0 to KL-1:
    IF k1[ i ] or *no writemask*:
        IF src2 is memory and evex.b == 1:
            t := src2.dword[0]
        ELSE:
            t := src2.dword[ i ]

        // FP32 FMA with daz in, ftz out and RNE rounding. MXCSR neither consulted nor updated.

        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+1]) * make_fp32(t.bfloat[1])
        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+0]) * make_fp32(t.bfloat[0])

    ELSE IF *zeroing*:
        srcdest.dword[ i ] := 0
    ELSE: // merge masking, dest element unchanged
        srcdest.dword[ i ] := origdest.dword[ i ]

srcdest[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VDPBF16PS __m128 _mm_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_mask_dpbf16_ps( __m128, __mmask8, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_maskz_dpbf16_ps(__mmask8, __m128, __m128bh, __m128bh);
VDPBF16PS __m256 _mm256_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_mask_dpbf16_ps(__m256, __mmask8, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_maskz_dpbf16_ps(__mmask8, __m256, __m256bh, __m256bh);
VDPBF16PS __m512 _mm512_dpbf16_ps(__m512, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_mask_dpbf16_ps(__m512, __mmask16, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_maskz_dpbf16_ps(__mmask16, __m512, __m512bh, __m512bh);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VEXPANDPD—Load Sparse Packed Double Precision Floating-Point Values From Dense Memory

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 88 /r VEXPANDPD xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 88 /r VEXPANDPD ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 88 /r VEXPANDPD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 8/4/2, contiguous, double precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] := SRC[k+63:k];
            k := k + 64
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXPANDPD __m512d _mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d _mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_mask_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d _mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_mask_expandloadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd( __mmask8 k, void * a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VEXPANDPS—Load Sparse Packed Single Precision Floating-Point Values From Dense Memory

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 16/8/4, contiguous, single precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := SRC[k+31:k];
            k := k + 32
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXPANDPS __m512 _mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 _mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 _mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 _mm256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 _mm_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 _mm_maskz_expandloadu_ps( __mmask8 k, void * a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4— Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 xmm1/m128, ymm2, imm8	A	V/V	AVX	Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 19 /r ib VEXTRACTF32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Extract 128 bits of packed single precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 19 /r ib VEXTRACTF32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 128 bits of packed single precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Extract 128 bits of packed double precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ OR AVX10.1	Extract 128 bits of packed double precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 1B /r ib VEXTRACTF32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ OR AVX10.1	Extract 256 bits of packed single precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 1B /r ib VEXTRACTF64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 256 bits of packed double precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 6 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTF32x4 (EVEX Encoded Versions) When Destination is a Register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;
FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTF32x4 (EVEX Encoded Versions) When Destination is Memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
```

```
            THEN DEST[i+31:i] := TMP_DEST[i+31:i]
            ELSE *DEST[i+31:i] remains unchanged*           ; merging-masking
    FI;
ENDFOR
```

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is a Register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is Memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
```

```
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VEXTRACTF32x8 (EVEX.U1.512 Encoded Version) When Destination is a Register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTF32x8 (EVEX.U1.512 Encoded Version) When Destination is Memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VEXTRACTF64x4 (EVEX.512 Encoded Version) When Destination is a Register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
```

```
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTF64x4 (EVEX.512 Encoded Version) When Destination is Memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE    ; merging-masking
            *DEST[i+63:i] remains unchanged*
    FI;
ENDFOR
```

VEXTRACTF128 (Memory Destination Form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
    1: DEST[127:0] := SRC1[255:128]
ESAC.
```

VEXTRACTF128 (Register Destination Form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
    1: DEST[127:0] := SRC1[255:128]
ESAC.
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_maskz_extractf32x4_ps( __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_extractf32x4_ps(__m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_maskz_extractf32x4_ps( __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_extractf32x8_ps(__m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_maskz_extractf32x8_ps( __mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_maskz_extractf64x2_pd( __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_extractf64x2_pd(__m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_maskz_extractf64x2_pd( __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_extractf64x4_pd( __m512d a, const int nidx);
```

```
VEXTRACTF64x4 __m256d _mm512_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_maskz_extractf64x4_pd( __mmask8 k, __m512d a, const int nidx);
VEXTRACTF128 __m128 _mm256_extractf128_ps (__m256 a, int offset);
VEXTRACTF128 __m128d _mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-56, "Type E6NF Class Exception Conditions."

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 xmm1/m128, ymm2, imm8	A	V/V	AVX2	Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 39 /r ib VEXTRACTI32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 39 /r ib VEXTRACTI32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ OR AVX10.1	Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 3B /r ib VEXTRACTI32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ OR AVX10.1	Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 3B /r ib VEXTRACTI64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F OR AVX10.1	Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTI32x4 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;
FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTI32x4 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
```

```
FI;

FOR j := 0 TO 3
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*            ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
        01: TMP_DEST[127:0] := SRC1[255:128]
        10: TMP_DEST[127:0] := SRC1[383:256]
        11: TMP_DEST[127:0] := SRC1[511:384]
    ESAC.
FI;

FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                 ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.
FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*   ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI128 (memory destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
    1: DEST[127:0] := SRC1[255:128]
ESAC.
```

VEXTRACTI128 (register destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
    1: DEST[127:0] := SRC1[255:128]
ESAC.
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_extracti32x4_epi32(__m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_extracti32x8_epi32(__m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_mask_extracti32x8_epi32(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_extracti64x2_epi64(__m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_maskz_extracti64x2_epi64(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_mask_extracti64x4_epi64(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI128 __m128i _mm256_extracti128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-56, “Type E6NF Class Exception Conditions.”

Additionally:

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VFCMADDCPH/VFMADDCPH—Complex Multiply and Accumulate FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP6.W0 56 /r VFCMADDCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP6.W0 56 /r VFCMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP6.W0 56 /r VFCMADDCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.
EVEX.128.F3.MAP6.W0 56 /r VFMADDCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP6.W0 56 /r VFMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and the complex conjugate of ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP6.W0 56 /r VFMADDCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and the complex conjugate of zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMADDCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp[2*i+1] := dest.fp16[2*i+1] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // conjugate version subtracts odd final term
        dest.fp16[2*i+0] := tmp[2*i+0] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp[2*i+1] - src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

VFMADDCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp[2*i+1] := dest.fp16[2*i+1] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // non-conjugate version subtracts even term
        dest.fp16[2*i+0] := tmp[2*i+0] - src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp[2*i+1] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMADDCPH __m128h _mm_fcmadd_pch (__m128h a, __m128h b, __m128h c);
VFCMADDCPH __m128h _mm_mask_fcmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCPH __m128h _mm_mask3_fcmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFCMADDCPH __m128h _mm_maskz_fcmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFCMADDCPH __m256h _mm256_fcmadd_pch (__m256h a, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask_fcmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask3_fcmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFCMADDCPH __m256h _mm256_maskz_fcmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFCMADDCPH __m512h _mm512_fcmadd_pch (__m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask_fcmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask3_fcmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFCMADDCPH __m512h _mm512_maskz_fcmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask_fcmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask3_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFCMADDCPH __m512h _mm512_maskz_fcmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFMADDCPH __m128h _mm_fmadd_pch (__m128h a, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask3_fmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFMADDCPH __m128h _mm_maskz_fmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFMADDCPH __m256h _mm256_fmadd_pch (__m256h a, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask_fmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask3_fmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFMADDCPH __m256h _mm256_maskz_fmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFMADDCPH __m512h _mm512_fmadd_pch (__m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask_fmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask3_fmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFMADDCPH __m512h _mm512_maskz_fmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_fmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask_fmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask3_fmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFMADDCPH __m512h _mm512_maskz_fmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMADDCSH/VFMADDCSH—Complex Multiply and Accumulate Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP6.W0 57 /r VFCMADDCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.MAP6.W0 57 /r VFMADDCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMADDCSH dest{k1}, src1, src2 (AVX512)

```
IF k1[0] or *no writemask*:
    tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
    tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]

    // conjugate version subtracts odd final term
    dest.fp16[0] := tmp[0] + src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp[1] - src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

```
VFMADDCSH dest{k1}, src1, src2 (AVX512)
IF k1[0] or *no writemask*:
    tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
    tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]

    // non-conjugate version subtracts last even term
    dest.fp16[0] := tmp[0] - src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp[1] + src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMADDCSH __m128h _mm_fcmadd_round_sch (__m128h a, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_mask_fcmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_mask3_fcmadd_round_sch (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
VFCMADDCSH __m128h _mm_maskz_fcmadd_round_sch (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
VFCMADDCSH __m128h _mm_fcmadd_sch (__m128h a, __m128h b, __m128h c);
VFCMADDCSH __m128h _mm_mask_fcmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCSH __m128h _mm_mask3_fcmadd_sch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFCMADDCSH __m128h _mm_maskz_fcmadd_sch (__mmask8 k, __m128h a, __m128h b, __m128h c);

VFMADDCSH __m128h _mm_fmadd_round_sch (__m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_mask_fmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_mask3_fmadd_round_sch (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
VFMADDCSH __m128h _mm_maskz_fmadd_round_sch (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCSH __m128h _mm_fmadd_sch (__m128h a, __m128h b, __m128h c);
VFMADDCSH __m128h _mm_mask_fmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCSH __m128h _mm_mask3_fmadd_sch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFMADDCSH __m128h _mm_maskz_fmadd_sch (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-60, “Type E10 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.MAP6.W0 D6 /r VFCMULCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.F2.MAP6.W0 D6 /r VFCMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.F2.MAP6.W0 D6 /r VFCMULCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.F3.MAP6.W0 D6 /r VFMULCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.F3.MAP6.W0 D6 /r VFMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Complex multiply a pair of FP16 values from ymm2 and the complex conjugate of ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.F3.MAP6.W0 D6 /r VFMULCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from zmm2 and the complex conjugate of zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMULCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256 or 512
KL := VL/32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // conjugate version subtracts odd final term
        dest.fp16[2*i] := tmp.fp16[2*i+0] +src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] - src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

VFMULCPH dest{k1}, src1, src2 (AVX512)

```
VL = 128, 256 or 512
KL := VL/32

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to kl-1:
    IF k1[i] or *no writemask*:
        tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        // non-conjugate version subtracts last even term
        dest.fp16[2*i+0] := tmp.fp16[2*i+0] - src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMULCPH __m128h _mm_cmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h _mm_mask_cmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h _mm_maskz_cmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h _mm256_cmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h _mm256_mask_cmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h _mm256_maskz_cmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h _mm512_cmul_pch (__m512h a, __m512h b);
VFCMULCPH __m512h _mm512_mask_cmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_maskz_cmul_pch (__mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_cmul_round_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_cmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_cmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m128h _mm_fcmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h _mm_mask_fcmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h _mm_maskz_fcmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h _mm256_fcmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h _mm256_mask_fcmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h _mm256_maskz_fcmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h _mm512_fcmul_pch (__m512h a, __m512h b);
VFCMULCPH __m512h _mm512_mask_fcmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_maskz_fcmul_pch (__mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_fcmul_round_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_fcmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_fcmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);

VFMULCPH __m128h _mm_fmul_pch (__m128h a, __m128h b);
VFMULCPH __m128h _mm_mask_fmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_maskz_fmul_pch (__mmask8 k, __m128h a, __m128h b);
VFMULCPH __m256h _mm256_fmul_pch (__m256h a, __m256h b);
VFMULCPH __m256h _mm256_mask_fmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_maskz_fmul_pch (__mmask8 k, __m256h a, __m256h b);
VFMULCPH __m512h _mm512_fmul_pch (__m512h a, __m512h b);
VFMULCPH __m512h _mm512_mask_fmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_maskz_fmul_pch (__mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_fmul_round_pch (__m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_mask_fmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_maskz_fmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m128h _mm_mask_mul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_maskz_mul_pch (__mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_mul_pch (__m128h a, __m128h b);
VFMULCPH __m256h _mm256_mask_mul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_maskz_mul_pch (__mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_mul_pch (__m256h a, __m256h b);
VFMULCPH __m512h _mm512_mask_mul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_maskz_mul_pch (__mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_mul_pch (__m512h a, __m512h b);
VFMULCPH __m512h _mm512_mask_mul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_maskz_mul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFMULCPH __m512h _mm512_mul_round_pch (__m512h a, __m512h b, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMULCSH/VFMULCSH—Complex Multiply Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.MAP6.W0 D7 /r VFCMULCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].
EVEX.LLIG.F3.MAP6.W0 D7 /r VFMULCSH xmm1{k1}{z}, xmm2, xmm3/m32 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Complex multiply a pair of FP16 values from xmm2 and the complex conjugate of xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMULCSH dest{k1}, src1, src2 (AVX512)

```
KL := VL / 32

IF k1[0] or *no writemask*:
    tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
    tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]

    // conjugate version subtracts odd final term
    dest.fp16[0] := tmp.fp16[0] + src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp.fp16[1] - src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

VFMULCSH dest{k1}, src1, src2 (AVX512)

```
KL := VL / 32

IF k1[0] or *no writemask*:
    // non-conjugate version subtracts last even term
    tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
    tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]
    dest.fp16[0] := tmp.fp16[0] - src1.fp16[1] * src2.fp16[1]
    dest.fp16[1] := tmp.fp16[1] + src1.fp16[0] * src2.fp16[1]
ELSE IF *zeroing*:
    dest.fp16[0] := 0
    dest.fp16[1] := 0

DEST[127:32] := src1[127:32] // copy upper part of src1
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFCMULCSH __m128h _mm_cmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_mask_cmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_maskz_cmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_cmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h _mm_mask_cmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_maskz_cmul_sch (__mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_fcmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_mask_fcmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_maskz_fcmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h _mm_fcmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h _mm_mask_fcmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h _mm_maskz_fcmul_sch (__mmask8 k, __m128h a, __m128h b);

VFMULCSH __m128h _mm_fmul_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mask_fmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_maskz_fmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_fmul_sch (__m128h a, __m128h b);
VFMULCSH __m128h _mm_mask_fmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_maskz_fmul_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_mask_mul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_maskz_mul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mul_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h _mm_mask_mul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_maskz_mul_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h _mm_mul_sch (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-60, “Type E10 Class Exception Conditions.”

Additionally:

#UD	If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 54 /r ib VFIXUPIMMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.
EVEX.256.66.0F3A.W1 54 /r ib VFIXUPIMMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.
EVEX.512.66.0F3A.W1 54 /r ib VFIXUPIMMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform fix-up of quad-word elements encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.

Operation

```
enum TOKEN_TYPE
```

```
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] := ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }    ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] := dest[63:0];          ; preserve content of DEST
        0001: dest[63:0] := tsrc[63:0];          ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] := QNaN(tsrc[63:0]);
        0011: dest[63:0] := QNAN_Indefinite;
        0100: dest[63:0] := -INF;
        0101: dest[63:0] := +INF;
        0110: dest[63:0] := tsrc.sign? –INF : +INF;
        0111: dest[63:0] := -0;
        1000: dest[63:0] := +0;
        1001: dest[63:0] := -1;
        1010: dest[63:0] := +1;
        1011: dest[63:0] := ½;
        1100: dest[63:0] := 90.0;
        1101: dest[63:0] := PI/2;
        1110: dest[63:0] := MAX_FLOAT;
        1111: dest[63:0] := -MAX_FLOAT;
    }        ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
            ; end fault reporting
    return dest[63:0];
}           ; end of FIXUPIMM_DP()
```

VFIXUPIMMPD

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])
                ELSE
                    DEST[i+63:i] := FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] := 0             ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Immediate Control Description:

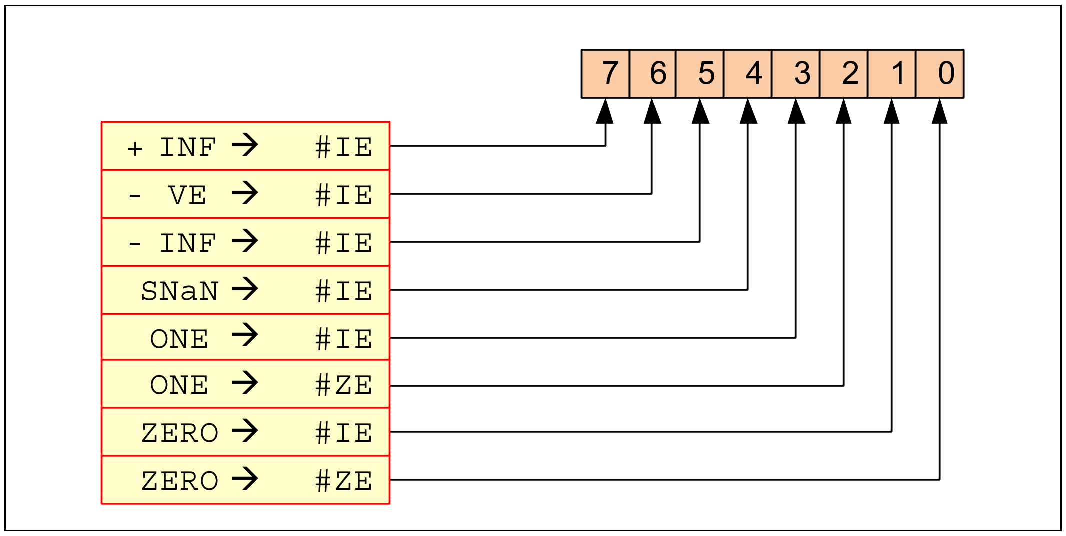


Figure 5-9. VFIXUPIMMPD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMPD __m512d _mm512_fixupimm_pd(__m512d a, __m512d b, __m512i c, int imm8);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d a, __mmask8 k, __m512d b, __m512i c, int imm8);

VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512d b, __m512i c, int imm8);
VFIXUPIMMPD __m512d _mm512_fixupimm_round_pd(__m512d a, __m512d b, __m512i c, int imm8, int sae);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d a, __mmask8 k, __m512d b, __m512i c, int imm8, int sae);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512d b, __m512i c, int imm8, int sae);
VFIXUPIMMPD __m256d _mm256_fixupimm_pd(__m256d a, m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d a, __mmask8 k, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_maskz_fixupimm_pd(__mmask8 k, __m256d a, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m128d _mm_fixupimm_pd(__m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_mask_fixupimm_pd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_maskz_fixupimm_pd(__mmask8 k, __m128d a, __m128d b, __m128i c, int imm8);

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 54 /r VFIXUPIMMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.
EVEX.256.66.0F3A.W0 54 /r VFIXUPIMMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.
EVEX.512.66.0F3A.W0 54 /r ib VFIXUPIMMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform fix-up of doubleword elements encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPS can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] := ((src1[30:23] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[31:0]
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] := dest[31:0];         ; preserve content of DEST
        0001: dest[31:0] := tsrc[31:0];         ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] := QNaN(tsrc[31:0]);
        0011: dest[31:0] := QNAN_Indefinite;
        0100: dest[31:0] := -INF;
        0101: dest[31:0] := +INF;
        0110: dest[31:0] := tsrc.sign? –INF : +INF;
        0111: dest[31:0] := -0;
        1000: dest[31:0] := +0;
        1001: dest[31:0] := -1;
        1010: dest[31:0] := +1;
        1011:  dest[31:0] := ½;
        1100: dest[31:0] := 90.0;
        1101: dest[31:0] := PI/2;
        1110: dest[31:0] := MAX_FLOAT;
        1111: dest[31:0] := -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[31:0];
}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMPS (EVEX)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
                ELSE
                    DEST[i+31:i] := FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8 [7:0])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] := 0         ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Immediate Control Description:

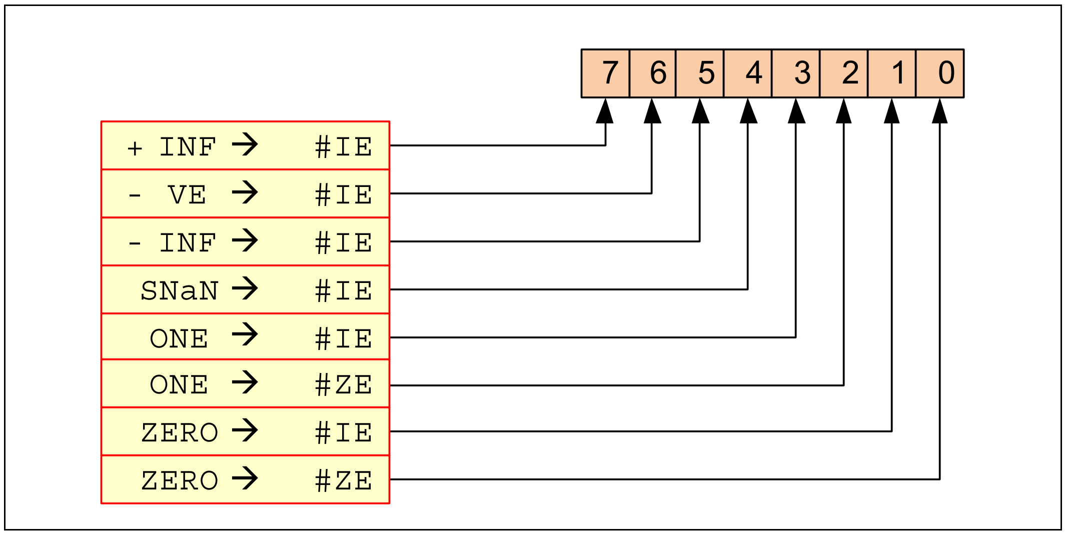


Figure 5-10. VFIXUPIMMPS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

```
VFIXUPIMMPS __m512 _mm512_fixupimm_ps( __m512 a, __m512 b, __m512i c, int imm8);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_ps(__m512 a, __mmask16 k, __m512 b, __m512i c, int imm8);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_ps( __mmask16 k, __m512 a, __m512 b, __m512i c, int imm8);
VFIXUPIMMPS __m512 _mm512_fixupimm_round_ps( __m512 a, __m512 b, __m512i c, int imm8, int sae);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_round_ps(__m512 a, __mmask16 k, __m512 b, __m512i c, int imm8, int sae);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_round_ps( __mmask16 k, __m512 a, __m512 b, __m512i c, int imm8, int sae);
VFIXUPIMMPS __m256 _mm256_fixupimm_ps( __m256 a, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m256 _mm256_mask_fixupimm_ps(__m256 a, __mmask8 k, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m256 _mm256_maskz_fixupimm_ps( __mmask8 k, __m256 a, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS __m128 _mm_fixupimm_ps( __m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 _mm_mask_fixupimm_ps(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 _mm_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128 b, __m128i c, int imm8);
```

SIMD Floating-Point Exceptions

Zero, Invalid.

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform a fix-up of the low quadword element encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64- bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] := ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] := dest[63:0]          ; preserve content of DEST
        0001: dest[63:0] := tsrc[63:0];         ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] := QNaN(tsrc[63:0]);
        0011: dest[63:0] := QNAN_Indefinite;
        0100:dest[63:0] := -INF;
        0101: dest[63:0] := +INF;
        0110: dest[63:0] := tsrc.sign? –INF : +INF;
        0111: dest[63:0] := -0;
        1000: dest[63:0] := +0;
        1001: dest[63:0] := -1;
        1010: dest[63:0] := +1;
        1011: dest[63:0] := ½;
        1100: dest[63:0] := 90.0;
        1101: dest[63:0] := PI/2;
        1110: dest[63:0] := MAX_FLOAT;
        1111: dest[63:0] := -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[63:0];
}       ; end of FIXUPIMM_DP()
```

VFIXUPIMMSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] := FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE  DEST[63:0] := 0            ; zeroing-masking
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Immediate Control Description:

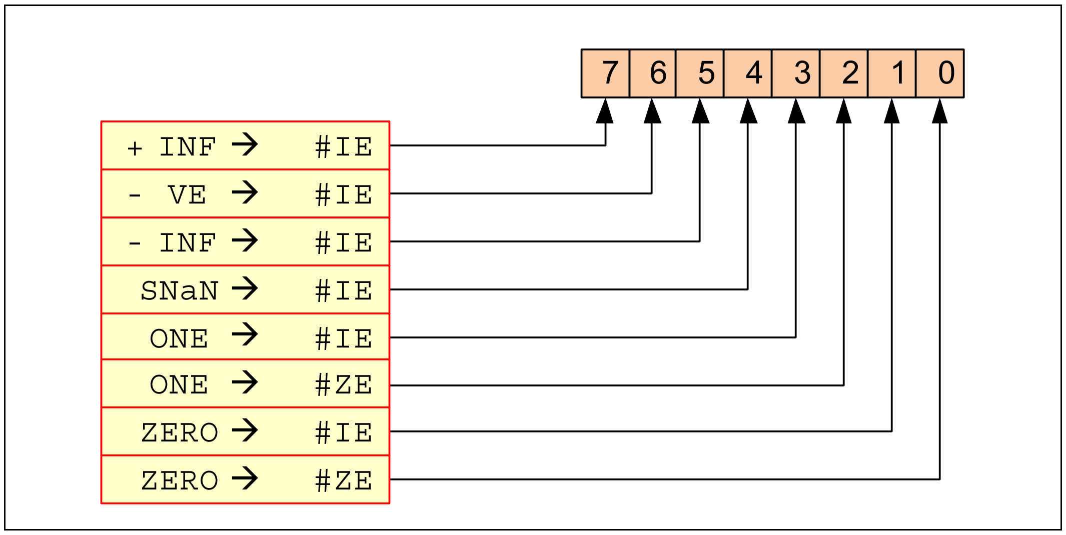


Figure 5-11. VFIXUPIMMSD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSD __m128d _mm_fixupimm_sd(__m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_sd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMSD __m128d _mm_fixupimm_round_sd(__m128d a, __m128d b, __m128i c, int imm8, int sae);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_round_sd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8, int sae);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128d b, __m128i c, int imm8, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Perform a fix-up of the low doubleword element encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_SP (dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] := ((src1[30:23] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] := dest[31:0];         ; preserve content of DEST
        0001: dest[31:0] := tsrc[31:0];         ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] := QNaN(tsrc[31:0]);
        0011: dest[31:0] := QNAN_Indefinite;
        0100: dest[31:0] := -INF;
        0101: dest[31:0] := +INF;
        0110: dest[31:0] := tsrc.sign? –INF : +INF;
        0111: dest[31:0] := -0;
        1000: dest[31:0] := +0;
        1001: dest[31:0] := -1;
        1010: dest[31:0] := +1;
        1011: dest[31:0] := ½;
        1100: dest[31:0] := 90.0;
        1101: dest[31:0] := PI/2;
        1110: dest[31:0] := MAX_FLOAT;
        1111: dest[31:0] := -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[31:0];
}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] := FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE  DEST[31:0] := 0               ; zeroing-masking
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Immediate Control Description:

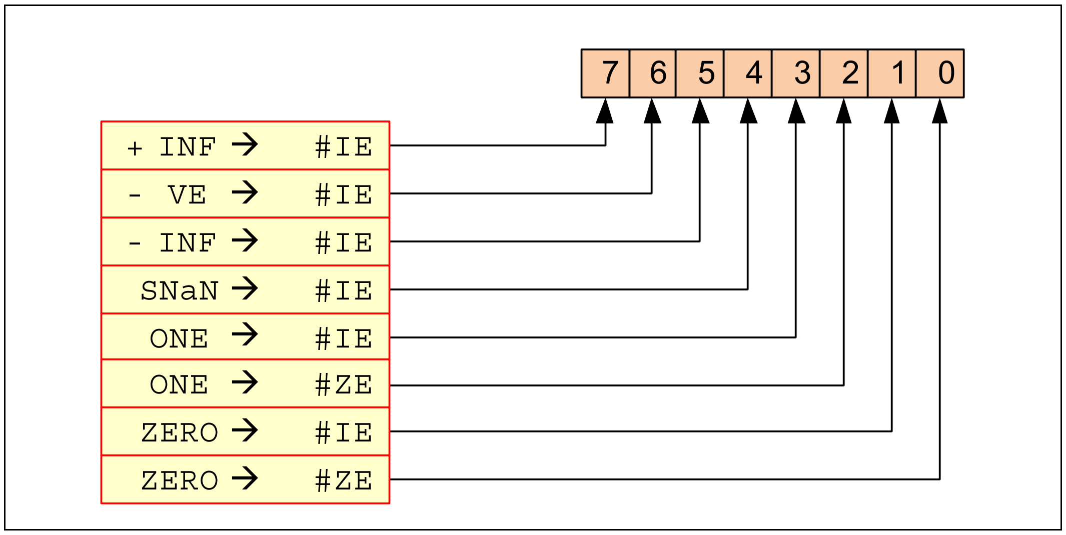


Figure 5-12. VFIXUPIMMSS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

```
VFIXUPIMMSS __m128 _mm_fixupimm_ss( __m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_ss(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_ss( __mmask8 k, __m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMSS __m128 _mm_fixupimm_round_ss( __m128 a, __m128 b, __m128i c, int imm8, int sae);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_round_ss(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8, int sae);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_round_ss( __mmask8 k, __m128 a, __m128 b, __m128i c, int imm8, int sae);
```

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.
EVEX.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-add computation on packed double precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VF[,N]MADD[132,213,231]PH—Fused Multiply-Add of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 98 /r VFMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 98 /r VFMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 98 /r VFMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 A8 /r VFMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, add to xmm3/m128/m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 A8 /r VFMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, add to ymm3/m256/m16bcst, and store the result in ymm1.
EVEX.512.66.MAP6.W0 A8 /r VFMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, add to zmm3/m512/m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 B8 /r VFMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 B8 /r VFMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 B8 /r VFMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add to zmm1, and store the result in zmm1.
EVEX.128.66.MAP6.W0 9C /r VFNMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 9C /r VFNMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 9C /r VFNMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 AC /r VFNMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m128/m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 AC /r VFNMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Add this value to ymm3/m256/m16bcst, and store the result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.MAP6.W0 AC /r VFNMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Add this value to zmm3/m512/m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 BC /r VFNMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 BC /r VFNMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 BC /r VFNMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add or negated multiply-add computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B + C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-5.

The destination elements are updated according to the writemask.

Table 5-5. VF[,N]MADD[132,213,231]PH Notation for Operands

Notation	Operands
132	dest = ± dest*src3+src2
231	dest = ± src2*src3+dest
213	dest = ± src2*dest+src3

Operation

VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 + SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 + DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADD132PH, VFMADD213PH , and VFMADD231PH:
__m128h _mm_fmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFNMADD132PH, VFNMADD213PH, and VFNMADD231PH:
__m128h _mm_fnmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fnmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fnmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fnmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fnmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fnmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fnmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fnmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fnmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fnmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fnmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 A8 /r VFMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 B8 /r VFMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 98 /r VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 A8 /r VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 B8 /r VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 98 /r VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 A8 /r VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 B8 /r VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1.
EVEX.512.66.0F38.W0 98 /r VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 A8 /r VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 B8 /r VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-add computation on packed single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
```

```
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
```

```
                IF (EVEX.b = 1)
                    THEN
                        DEST[i+31:i] :=
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                    ELSE
                        DEST[i+31:i] :=
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                FI;
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxPS __m512 _mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 _mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
VEX.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 99 /r VFMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-add computation on the low double precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

VFMADD132SD: Multiplies the low double precision floating-point value from the first source operand to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point values in the second source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low double precision floating-point value from the second source operand to the low double precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the third source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low double precision floating-point value from the second source to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the first source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN     DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := MAXVL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0
```

VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0
```

VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxSD __m128d _mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VF[,N]MADD[132,213,231]SH—Fused Multiply-Add of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 99 /r VFMADD132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, add to xmm2, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 A9 /r VFMADD213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, add to xmm3/m16, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 B9 /r VFMADD231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, add to xmm1, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 9D /r VFNMADD132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Add this value to xmm2, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 AD /r VFNMADD213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m16, and store the result in xmm1.
EVEX.LLIG.66.MAP6.W0 BD /r VFNMADD231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Add this value to xmm1, and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a scalar multiply-add or negated multiply-add computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation’ “132”, “213” and “231” indicate the use of the operands in ±A * B + C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-6.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 5-6. VF[,N]MADD[132,213,231]SH Notation for Operands

Notation	Operands
132	dest = ± dest*src3+src2
231	dest = ± src2*src3+dest
213	dest = ± src2*dest+src3

Operation

VF[,N]MADD132SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MADD213SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MADD231SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SH, VFMADD213SH, and VFMADD231SH:

```
__m128h _mm_fmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

VFNMADD132SH, VFNMADD213SH, and VFNMADD231SH:

```
__m128h _mm_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-add computation on single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

VFMADD132SS: Multiplies the low single precision floating-point value from the first source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the second source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the third source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the first source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0]] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192]
FI
```

VFMADDSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192]
FI
```

VFMADDSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192]
FI
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                FI;
            FI

        ELSE
            IF *merging-masking*              ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking*              ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
            THEN DEST[i+63:i] :=
                RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            ELSE DEST[i+63:i] :=
                RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        FI
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] := 0
        FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDSUBxxxPD __m512d _mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d _mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d _mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d _mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d _mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH—Fused Multiply-Alternating Add/Subtract of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 96 /r VFMADDSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add/subtract elements in xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 96 /r VFMADDSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add/subtract elements in ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 96 /r VFMADDSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add/subtract elements in zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 A6 /r VFMADDSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 A6 /r VFMADDSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 A6 /r VFMADDSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 B6 /r VFMADDSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add/subtract elements in xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 B6 /r VFMADDSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add/subtract elements in ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 B6 /r VFMADDSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add/subtract elements in zmm1, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add (odd elements) or multiply-subtract (even elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation' "132", "213" and "231" indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-10.

The destination elements are updated according to the writemask.

Table 5-7. VFMADDSUB[132,213,231]PH Notation for Odd and Even Elements

Notation	Odd Elements	Even Elements
132	dest = dest*src3+src2	dest = dest*src3-src2
231	dest = src2*src3+dest	dest = src2*src3-dest
213	dest = src2*dest+src3	dest = src2*dest-src3

Operation

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3)
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PH, VFMADDSUB213PH, and VFMADDSUB231PH:

```
__m128h _mm_fmaddsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmaddsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmaddsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmaddsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmaddsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmaddsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmaddsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmaddsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmaddsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmaddsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmaddsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmaddsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmaddsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmaddsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in zmm2 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 A6 /r VFMADDSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 B6 /r VFMADDSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 96 /r VFMADDSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

```
VFMADDSUB132PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM :=2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMADDSUB213PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMADDSUB231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] :=RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                    FI;
                FI

        ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
```

```
                        ELSE
                              DEST[i+31:i] :=
                  RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                  FI;
            ELSE
                  IF (EVEX.b = 1)
                        THEN
                              DEST[i+31:i] :=
                  RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                        ELSE
                              DEST[i+31:i] :=
                  RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                  FI;
      FI
   ELSE
      IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                              ; zeroing-masking
                  DEST[i+31:i] := 0
      FI
 FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
 THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
 ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
 i := j * 32
 IF k1[j] OR *no writemask*
      THEN
            IF j *is even*
                  THEN DEST[i+31:i] :=
                        RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                  ELSE DEST[i+31:i] :=
                        RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI
      ELSE
            IF *merging-masking*              ; merging-masking
                  THEN *DEST[i+31:i] remains unchanged*
                  ELSE                        ; zeroing-masking
                        DEST[i+31:i] := 0
            FI
 FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
```

```
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMADDSUBxxxPS __m512 _mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256 _mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 _mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 _mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9A /r VFMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 AA /r VFMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 BA /r VFMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 9A /r VFMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 AA /r VFMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 BA /r VFMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.S
EVEX.128.66.0F38.W1 9A /r VFMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 AA /r VFMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 BA /r VFMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 9A /r VFMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 AA /r VFMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 BA /r VFMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9A /r VFMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 AA /r VFMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 BA /r VFMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-subtract computation on packed double precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3 (VEX Encoded Versions)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
```

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
            ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
+31:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX Encoded Versions, When SRC3 Operand is a Memory Source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VF[,N]MSUB[132,213,231]PH—Fused Multiply-Subtract of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 9A /r VFMSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 9A /r VFMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 9A /r VFMSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 AA /r VFMSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 AA /r VFMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 AA /r VFMSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 BA /r VFMSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 BA /r VFMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 BA /r VFMSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract zmm1, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 9E /r VFNMSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 9E /r VFNMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Subtract ymm2 from this value, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 9E /r VFNMSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Subtract zmm2 from this value, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 AE /r VFNMSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m128/m16bcst from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 AE /r VFNMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Subtract ymm3/m256/m16bcst from this value, and store the result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.MAP6.W0 AE /r VFNMSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Subtract zmm3/m512/m16bcst from this value, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 BE /r VFNMSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 BE /r VFNMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Subtract ymm1 from this value, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 BE /r VFNMSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Subtract zmm1 from this value, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-subtract or a negated multiply-subtract computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B − C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-8.

The destination elements are updated according to the writemask.

Table 5-8. VF[,N]MSUB[132,213,231]PH Notation for Operands

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Operation

VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 - SRC2.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] - t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 - DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUB132PH, VFMSUB213PH, and VFMSUB231PH:
__m128h _mm_fmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFNMSUB132PH, VFNMSUB213PH, and VFNMSUB231PH:
__m128h _mm_fnmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fnmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fnmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fnmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fnmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fnmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fnmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fnmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fnmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fnmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fnmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.
EVEX.512.66.0F38.W0 9A /r VFMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 AA /r VFMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 BA /r VFMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a set of SIMD multiply-subtract computation on packed single precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPS __m512 _mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 _mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
VEX.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 9B /r VFMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AB /r VFMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BB /r VFMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-subtract computation on the low packed double precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

VFMSUB132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VF[,N]MSUB[132,213,231]SH—Fused Multiply-Subtract of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 9B /r VFMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, subtract xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AB /r VFMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, subtract xmm3/m16, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BB /r VFMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, subtract xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 9F /r VFNMSUB132SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 AF /r VFNMSUB213SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m16 from this value, and store the result in xmm1 subject to writemask k1.
EVEX.LLIG.66.MAP6.W0 BF /r VFNMSUB231SH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a scalar multiply-subtract or negated multiply-subtract computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation' "132", "213" and "231" indicate the use of the operands in ±A * B − C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-9.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 5-9. VF[,N]MSUB[132,213,231]SH Notation for Operands

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Operation

VF[,N]MSUB132SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB213SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

VF[,N]MSUB231SH DEST, SRC2, SRC3 (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC3 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    IF *negative form*:
        DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
    ELSE:
        DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SH, VFMSUB213SH, and VFMSUB231SH:

```
__m128h _mm_fmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

VFNMSUB132SH, VFNMSUB213SH, and VFNMSUB231SH:

```
__m128h _mm_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 9B /r VFMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AB /r VFMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BB /r VFMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD multiply-subtract computation on the low packed single precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

VFMSUB132SS: Multiplies the low packed single precision floating-point value from the first source operand to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the third source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single precision floating-point value from the second source to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the first source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 97 /r VFMSUBADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 A7 /r VFMSUBADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W1 B7 /r VFMSUBADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMSUBADD132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192]
FI
```

VFMSUBADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192]
FI
```

VFMSUBADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192]
FI
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
```

```
            THEN
                IF j *is even*
                    THEN DEST[i+63:i] :=
                        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                    ELSE DEST[i+63:i] :=
                        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                FI
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                    FI;
            FI

        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN
         IF j *is even*
            THEN DEST[i+63:i] :=
               RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            ELSE DEST[i+63:i] :=
               RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
         FI
      ELSE
         IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                 ; zeroing-masking
               DEST[i+63:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN
         IF j *is even*
            THEN
               IF (EVEX.b = 1)
                  THEN
                     DEST[i+63:i] :=
               RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                  ELSE
                     DEST[i+63:i] :=
               RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
               FI;
            ELSE
               IF (EVEX.b = 1)
                  THEN
                     DEST[i+63:i] :=
               RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                  ELSE
                     DEST[i+63:i] :=
               RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
```

```
                        FI;
                FI
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])

                        ELSE
                            DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBADDxxxPD __m512d _mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d _mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d _mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH—Fused Multiply-Alternating Subtract/Add of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 97 /r VFMSUBADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract/add elements in xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 97 /r VFMSUBADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract/add elements in ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 97 /r VFMSUBADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract/add elements in zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 A7 /r VFMSUBADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 A7 /r VFMSUBADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 A7 /r VFMSUBADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 B7 /r VFMSUBADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract/add elements in xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 B7 /r VFMSUBADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract/add elements in ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 B7 /r VFMSUBADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract/add elements in zmm1, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a packed multiply-add (even elements) or multiply-subtract (odd elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation "132", "213" and "231" indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-10.

The destination elements are updated according to the writemask.

Table 5-10. VFMSUBADD[132,213,231]PH Notation for Odd and Even Elements

Notation	Odd Elements	Even Elements
132	dest = dest*src3-src2	dest = dest*src3+src2
231	dest = src2*src3-dest	dest = src2*src3+dest
213	dest = src2*dest-src3	dest = src2*dest+src3

Operation

VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
```

```
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0:
```

VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3 )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0:
```

VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j] )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j] )
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBADD132PH, VFMSUBADD213PH, and VFMSUBADD231PH:
__m128h _mm_fmsubadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsubadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsubadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsubadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmsubadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmsubadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmsubadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmsubadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmsubadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmsubadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmsubadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmsubadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmsubadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmsubadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 A7 /r VFMSUBADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.512.66.0F38.W0 B7 /r VFMSUBADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
       THEN
           IF j *is even*
               THEN DEST[i+31:i] :=
                   RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
               ELSE DEST[i+31:i] :=
                   RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
           FI
       ELSE
           IF *merging-masking*                    ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE                                ; zeroing-masking
                   DEST[i+31:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
       THEN
           IF j *is even*
               THEN
                   IF (EVEX.b = 1)
                       THEN
                           DEST[i+31:i] :=
                       RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                       ELSE
                           DEST[i+31:i] :=
                       RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                   FI;
               ELSE
                   IF (EVEX.b = 1)
                       THEN
                           DEST[i+31:i] :=
                   RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                       ELSE
                           DEST[i+31:i] :=
                   RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
```

```
                    FI;
                FI

            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                    ELSE
                        DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                    FI;
            FI
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
```

```
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
       THEN
           IF j *is even*
               THEN
                   IF (EVEX.b = 1)
                       THEN
                           DEST[i+31:i] :=
                       RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                       ELSE
                           DEST[i+31:i] :=
                       RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                   FI;
               ELSE
                   IF (EVEX.b = 1)
                       THEN
                           DEST[i+31:i] :=
                   RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                       ELSE
                           DEST[i+31:i] :=
                   RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                   FI;
           FI
       ELSE
           IF *merging-masking*                    ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE                                ; zeroing-masking
                   DEST[i+31:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBADDxxxPS __m512 _mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBADDxxxPS __m256 _mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 9C /r VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W1 AC /r VFNMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.128.66.0F38.W1 BC /r VFNMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W1 9C /r VFNMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W1 AC /r VFNMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.256.66.0F38.W1 BC /r VFNMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9C /r VFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W1 AC /r VFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.512.66.0F38.W1 BC /r VFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 AC /r VFNMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 BC /r VFNMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 9C /r VFNMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 AC /r VFNMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 9C /r VFNMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 AC /r VFNMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 BC /r VFNMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 9C /r VFNMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 AC /r VFNMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 BC /r VFNMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 9C /r VFNMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 AC /r VFNMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 BC /r VFNMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] :=
         RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
      ELSE
         IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
               DEST[i+31:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1)
            THEN
               DEST[i+31:i] :=
         RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
            ELSE
               DEST[i+31:i] :=
         RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
         FI;

      ELSE
         IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
               DEST[i+31:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])

                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMADDxxxPS __m512 _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMADDxxxPS __m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 9D /r VFNMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AD /r VFNMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BD /r VFNMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
   THEN    DEST[63:0] := RoundFPControl(-(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
   ELSE
       IF *merging-masking*                ; merging-masking
           THEN *DEST[63:0] remains unchanged*
           ELSE                            ; zeroing-masking
               THEN DEST[63:0] := 0
       FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
   THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
   ELSE
       IF *merging-masking*                ; merging-masking
           THEN *DEST[63:0] remains unchanged*
           ELSE                            ; zeroing-masking
               THEN DEST[63:0] := 0
       FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMADDxxxSD __m128d _mm_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9D /r VFNMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 AD /r VFNMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 BD /r VFNMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 9D /r VFNMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AD /r VFNMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BD /r VFNMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMADD132SS: Multiplies the low packed single precision floating-point value from the first source operand to the low packed single precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single precision floating-point value in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single precision floating-point value in the third source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single precision floating-point value in the first source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.
EVEX.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.
EVEX.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 9E /r VFNMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.512.66.0F38.W1 AE /r VFNMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.
EVEX.512.66.0F38.W1 BE /r VFNMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPD __m512d _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMSUBxxxPD __m256d _mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.256.66.0F38.0 BE /r VFNMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.
EVEX.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.256.66.0F38.W0 BE /r VFNMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 9E /r VFNMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.512.66.0F38.W0 AE /r VFNMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.512.66.0F38.W0 BE /r VFNMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F OR AVX10.1	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source to the four, eight or sixteen packed single precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMSUBxxxPS __m512 _mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMSUBxxxPS __m256 _mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.
EVEX.LLIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMSUBxxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
VEX.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
EVEX.LLIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F OR AVX10.1	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

VFNMSUB132SS: Multiplies the low packed single precision floating-point value from the first source operand to the low packed single precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single precision floating-point value in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single precision floating-point value in the third source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single precision floating-point value from the second source to the low packed single precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single precision floating-point value in the first source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
   THEN     DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
   ELSE
       IF *merging-masking*                  ; merging-masking
           THEN *DEST[31:0] remains unchanged*
           ELSE                              ; zeroing-masking
               THEN DEST[31:0] := 0
       FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
   THEN     DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
   ELSE
       IF *merging-masking*                  ; merging-masking
           THEN *DEST[31:0] remains unchanged*
           ELSE                              ; zeroing-masking
               THEN DEST[31:0] := 0
       FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMSUBxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

VFPCLASSPD—Tests Types of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSPD instruction checks the packed double precision floating-point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

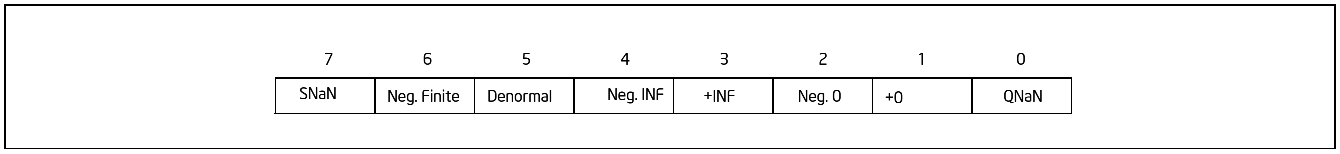


Figure 5-13. Imm8 Byte Specifier of Special Case Floating-Point Values for VFPCLASSPD/SD/PS/SS

Table 5-11. Classifier Operations for VFPCLASSPD/SD/PS/SS

Bits	Imm8[0]	Imm8[1]	Imm8[2]	Imm8[3]	Imm8[4]	Imm8[5]	Imm8[6]	Imm8[7]
Category	QNAN	PosZero	NegZero	PosINF	NegINF	Denormal	Negative	SNAN
Classifier	Checks for QNaN	Checks for +0	Checks for -0	Checks for +INF	Checks for -INF	Checks for Denormal	Checks for Negative finite	Checks for SNaN

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber := ExpAllZeros AND MantAllZeros
    SignalingBit := tsrc[51];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
                ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
                ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
                ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckFPClassDP() *//
```

VFPCLASSPD (EVEX Encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] := CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] := CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
            FI;
        ELSE DEST[j] := 0                  ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFPCLASSPD __mmask8 _mm512_fpclass_pd_mask( __m512d a, int c);
VFPCLASSPD __mmask8 _mm512_mask_fpclass_pd_mask( __mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 _mm256_fpclass_pd_mask( __m256d a, int c)
VFPCLASSPD __mmask8 _mm256_mask_fpclass_pd_mask( __mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 _mm_fpclass_pd_mask( __m128d a, int c)
VFPCLASSPD __mmask8 _mm_mask_fpclass_pd_mask( __mmask8 m, __m128d a, int c)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSPH—Test Types of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, xmm1/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, ymm1/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{k2}, zmm1/m512/m16bcst, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction checks the packed FP16 values in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 5-12 for the categories. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the corresponding bits in the destination mask register according to the writemask.

Table 5-12. Classifier Operations for VFPCLASSPH/VFPCLASSSH

Bits	Category	Classifier
imm8[0]	QNAN	Checks for QNAN
imm8[1]	PosZero	Checks +0
imm8[2]	NegZero	Checks for -0
imm8[3]	PosINF	Checks for $+\infty$
imm8[4]	NegINF	Checks for $-\infty$
imm8[5]	Denormal	Checks for Denormal
imm8[6]	Negative	Checks for Negative finite
imm8[7]	SNAN	Checks for SNAN

Operation

```
def check_fp_class_fp16(tsrc, imm8):
    negative := tsrc[15]
    exponent_all_ones := (tsrc[14:10] == 0x1F)
    exponent_all_zeros := (tsrc[14:10] == 0)
    mantissa_all_zeros := (tsrc[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := tsrc[9]

    snan := exponent_all_ones and not(mantissa_all_zeros) and not(signaling_bit)
    qnan := exponent_all_ones and not(mantissa_all_zeros) and signaling_bit
    positive_zero := not(negative) and zero
    negative_zero := negative and zero
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    negative_infinity := negative and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)
    finite_negative := negative and not(exponent_all_ones) and not(zero)

    return (imm8[0] and qnan) OR
        (imm8[1] and positive_zero) OR
        (imm8[2] and negative_zero) OR
        (imm8[3] and positive_infinity) OR
        (imm8[4] and negative_infinity) OR
        (imm8[5] and denormal) OR
        (imm8[6] and finite_negative) OR
        (imm8[7] and snan)
```

VFPCLASSPH dest{k2}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k2[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := SRC.fp16[0]
        ELSE:
            tsrc := SRC.fp16[i]
        DEST.bit[i] := check_fp_class_fp16(tsrc, imm8)
    ELSE:
        DEST.bit[i] := 0

DEST[MAXKL-1:kl] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFPCLASSPH __mmask8 _mm_fpclass_ph_mask (__m128h a, int imm8);
VFPCLASSPH __mmask8 _mm_mask_fpclass_ph_mask (__mmask8 k1, __m128h a, int imm8);
VFPCLASSPH __mmask16 _mm256_fpclass_ph_mask (__m256h a, int imm8);
VFPCLASSPH __mmask16 _mm256_mask_fpclass_ph_mask (__mmask16 k1, __m256h a, int imm8);
VFPCLASSPH __mmask32 _mm512_fpclass_ph_mask (__m512h a, int imm8);
VFPCLASSPH __mmask32 _mm512_mask_fpclass_ph_mask (__mmask32 k1, __m512h a, int imm8);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VFPCLASSPS—Tests Types of Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSPS instruction checks the packed single precision floating-point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];
    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
                ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
                ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
                ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckSPClassSP() *//
```

VFPCLASSPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] := CheckFPClassDP(SRC1[31:0], imm8[7:0]);
                ELSE
                    DEST[j] := CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
            FI;
        ELSE  DEST[j] := 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFPCLASSPS __mmask16 _mm512_fpclass_ps_mask( __m512 a, int c);
VFPCLASSPS __mmask16 _mm512_mask_fpclass_ps_mask( __mmask16 m, __m512 a, int c)
VFPCLASSPS __mmask8 _mm256_fpclass_ps_mask( __m256 a, int c)
VFPCLASSPS __mmask8 _mm256_mask_fpclass_ps_mask( __mmask8 m, __m256 a, int c)
VFPCLASSPS __mmask8 _mm_fpclass_ps_mask( __m128 a, int c)
VFPCLASSPS __mmask8 _mm_mask_fpclass_ps_mask( __mmask8 m, __m128 a, int c)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSD—Tests Type of a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 67 /r ib VFPCLASSSD k2 {k1}, xmm2/m64, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSSD instruction checks the low double precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    NegNum  := tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber := ExpAllZeros AND MantAllZeros
    SignalingBit := tsrc[51];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
            ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
            ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
            ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckFPClassDP() *//
```

VFPCLASSSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
            CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE  DEST[0] := 0                    ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSD __mmask8 _mm_fpclass_sd_mask(__m128d a, int c)

VFPCLASSSD __mmask8 _mm_mask_fpclass_sd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSH—Test Types of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 67 /r /ib VFPCLASSSH k1{k2}, xmm1/m16, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction checks the low FP16 value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 5-12 for the categories. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in the destination mask register according to the writemask. The other bits in the destination mask register are zeroed.

Operation

VFPCLASSSH dest{k2}, src, imm8

```
IF k2[0] or *no writemask*:
    DEST.bit[0] := check_fp_class_fp16(src.fp16[0], imm8)    // see VFPCLASSPH
ELSE:
    DEST.bit[0] := 0

DEST[MAXKL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSH __mmask8 _mm_fpclass_sh_mask (__m128h a, int imm8);
VFPCLASSSH __mmask8 _mm_mask_fpclass_sh_mask (__mmask8 k1, __m128h a, int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-60, “Type E10 Class Exception Conditions.”

VFPCLASSSS—Tests Type of a Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 67 /r VFPCLASSSS k2 {k1}, xmm2/m32, imm8	A	V/V	AVX512DQ OR AVX10.1	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The FPCLASSSS instruction checks the low single precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-11.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  := tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
         MantAllZeros := 1;
    ELSIF (tsrc[22:0]=0h) Then
         MantAllZeros := 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
                ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
                ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
                ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckSPClassSP() *//
```

VFPCLASSSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
            CheckFPClassSP(SRC1[31:0], imm8[7:0])
    ELSE  DEST[0] := 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSS __mmask8 _mm_fpclass_ss_mask(__m128 a, int c)
VFPCLASSSS __mmask8 _mm_mask_fpclass_ss_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) R AVX10.1[1]	Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32y	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of single precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VGATHERDPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        THEN DEST[i+31:i] :=
            MEM[BASE_ADDR +
                    SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

VGATHERDPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        THEN DEST[i+63:i] := MEM[BASE_ADDR +
                    SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD __m512d _mm512_i32gather_pd(__m256i vdx, void * base, int scale);

```
VGATHERDPD __m512d _mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d _mm256_mmask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d _mm_mmask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_i32gather_ps( __m512i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 _mm256_mmask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 _mm_mmask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of 8 single precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VGATHERQPS (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1

    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0
```

VGATHERQPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERQPD __m512d _mm512_i64gather_pd(__m512i vdx, void * base, int scale);

```
VGATHERQPD __m512d _mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d _mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d _mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_i64gather_ps( __m512i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 42 /r VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F OR AVX10.1	Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the biased exponents from the normalized double precision floating-point representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double precision floating-point value and written to the corresponding qword elements of the destination operand (the first operand) as double precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-13.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for the greatest integer not exceeding real number x.

Table 5-13. VGETEXPPD/SD Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN) then #IE If (SRC = denormal) then #DE
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Operation

```
NormalizeExpTinyDPFP(SRC[63:0])
{
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[51];              // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1 ;        // One bit shift left
        Dst.exp-- ;                 // Decrement the exponent
    }
    Dst.fraction := 0;              // zero out fraction bits
    Dst.sign := 1;                  // Return negative sign
    TMP[63:0] := MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction) ;
    Return (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign := 0;                  // Zero out sign bit
    Src.exp := SRC[62:52];
    Src.fraction := SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE            // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[63:0] := NormalizeExpTinyDPFP(SRC[63:0]) ;      // Get Normalized Exponent
            Set #DE
        }
        ELSE            // exponent value is correct
        {
            TMP[63:0] := (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction) ;
        }
        TMP := SAR(TMP, 52) ;           // Shift Arithmetic Right
        TMP := TMP – 1023;              // Subtract Bias
        Return CvtI2D(TMP);             // Convert INT to double precision floating-point number
    }
}
```

VGETEXPPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+63:i] :=
            ConvertExpDPFP(SRC[63:0])
                ELSE
                    DEST[i+63:i] :=
            ConvertExpDPFP(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd( __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd( __mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 42 /r VGETEXPPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.
EVEX.256.66.MAP6.W0 42 /r VGETEXPPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.
EVEX.512.66.MAP6.W0 42 /r VGETEXPPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction extracts the biased exponents from the normalized FP16 representation of each word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to an FP16 value and written to the corresponding word elements of the destination operand (the first operand) as FP16 numbers.

The destination elements are updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPH). Thus, the VGETEXPPH instruction does not require software to handle SIMD floating-point exceptions.

Table 5-14. VGETEXPPH/VGETEXPSH Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN), then #IE. If (SRC = denormal), then #DE.
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Operation

```
def normalize_exponent_tiny_fp16(src):
    jbit := 0
    // src & dst are FP16 numbers with sign(1b), exp(5b) and fraction (10b) fields
    dst.exp := 1                    // write bits 14:10
    dst.fraction := src.fraction // copy bits 9:0
    while jbit == 0:
        jbit := dst.fraction[9]     // msb of the fraction
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    dst.fraction := 0
    return dst

def getexp_fp16(src):
    src.sign := 0                   // make positive
    exponent_all_ones := (src[14:10] == 0x1F)
    exponent_all_zeros := (src[14:10] == 0)
    mantissa_all_zeros := (src[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := src[9]

    nan := exponent_all_ones and not(mantissa_all_zeros)
    snan := nan and not(signaling_bit)
    qnan := nan and signaling_bit
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)

    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)            // convert snan to a qnan
    if positive_infinity:
        return src
    if zero:
        return -INF
    if denormal:
        tmp := normalize_exponent_tiny_fp16(src)
        MXCSR.DE := 1
    else:
        tmp := src
    tmp := SAR(tmp, 10)             // shift arithmetic right
    tmp := tmp - 15                 // subtract bias
    return convert_integer_to_fp16(tmp)
```

VGETEXPPH dest{k1}, src

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := getexp_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPPH __m128h _mm_getexp_ph (__m128h a);
VGETEXPPH __m128h _mm_mask_getexp_ph (__m128h src, __mmask8 k, __m128h a);
VGETEXPPH __m128h _mm_maskz_getexp_ph (__mmask8 k, __m128h a);
VGETEXPPH __m256h _mm256_getexp_ph (__m256h a);
VGETEXPPH __m256h _mm256_mask_getexp_ph (__m256h src, __mmask16 k, __m256h a);
VGETEXPPH __m256h _mm256_maskz_getexp_ph (__mmask16 k, __m256h a);
VGETEXPPH __m512h _mm512_getexp_ph (__m512h a);
VGETEXPPH __m512h _mm512_mask_getexp_ph (__m512h src, __mmask32 k, __m512h a);
VGETEXPPH __m512h _mm512_maskz_getexp_ph (__mmask32 k, __m512h a);
VGETEXPPH __m512h _mm512_getexp_round_ph (__m512h a, const int sae);
VGETEXPPH __m512h _mm512_mask_getexp_round_ph (__m512h src, __mmask32 k, __m512h a, const int sae);
VGETEXPPH __m512h _mm512_maskz_getexp_round_ph (__mmask32 k, __m512h a, const int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Values to Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F OR AVX10.1	Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Extracts the biased exponents from the normalized single precision floating-point representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single precision floating-point value and written to the corresponding dword elements of the destination operand (the first operand) as single precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

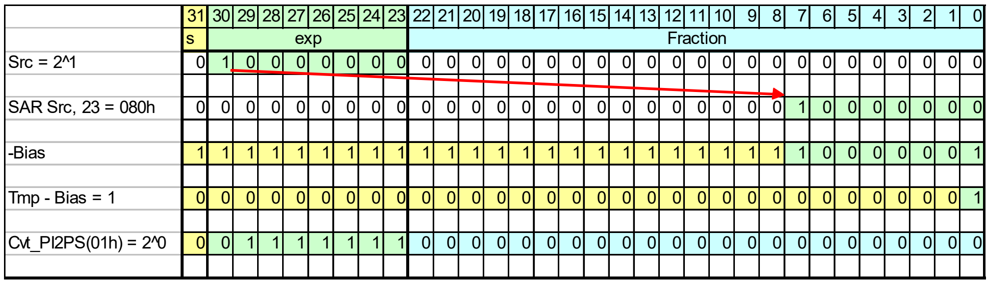
Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

Table 5-15. VGETEXPPS/SS Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN) then #IE If (SRC = denormal) then #DE
0 < |src1| < INF	floor(log$_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	s	exp								Fraction																						
Src = 2^1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SAR Src, 23 = 080h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
-Bias	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
Tmp - Bias = 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Cvt_PI2PS(01h) = 2^0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5-14. VGETEXPPS Functionality On Normal Input values

Operation

```
NormalizeExpTinySPFP(SRC[31:0])
{
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[22];           // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1 ;     // One bit shift left
        Dst.exp-- ;                 // Decrement the exponent
    }
    Dst.fraction := 0;              // zero out fraction bits
    Dst.sign := 1;                  // Return negative sign
    TMP[31:0] := MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction) ;
    Return (TMP[31:0]);
}
ConvertExpSPFP(SRC[31:0])
{
    Src.sign := 0;                  // Zero out sign bit
    Src.exp := SRC[30:23];
    Src.fraction := SRC[22:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
```

```
            Return QNAN(SRC);
        }
        // Check for +INF
        IF (Src = +INF) RETURN (Src);

        // check if zero operand
        IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
        }
        ELSE            // check if denormal operand (notice that MXCSR.DAZ = 0)
        {
            IF ((Src.exp = 0) AND (Src.fraction != 0))
            {
                TMP[31:0] := NormalizeExpTinySPFP(SRC[31:0]) ;      // Get Normalized Exponent
                Set #DE
            }
            ELSE            // exponent value is correct
            {
                TMP[31:0] := (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction) ;
            }
            TMP := SAR(TMP, 23) ;           // Shift Arithmetic Right
            TMP := TMP – 127;               // Subtract Bias
            Return CvtI2S(TMP);             // Convert INT to single precision floating-point number
        }
}
```

VGETEXPPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+31:i] :=
            ConvertExpSPFP(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
            ConvertExpSPFP(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPS __m512 _mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 _mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_maskz_getexp_ps(__mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_getexp_round_ps(__m512 a, int sae);
VGETEXPPS __m512 _mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 _mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 _mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 _mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 _mm256_maskz_getexp_ps(__mmask8 k, __m256 a);
VGETEXPPS __m128 _mm_getexp_ps(__m128 a);
VGETEXPPS __m128 _mm_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a);
VGETEXPPS __m128 _mm_maskz_getexp_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETEXPSD—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	A	V/V	AVX512F OR AVX10.1	Convert the biased exponent (bits 62:52) of the low double precision floating-point value in xmm3/m64 to a double precision floating-point value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Extracts the biased exponent from the normalized double precision floating-point representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double precision floating-point value and written to the destination operand (the first operand) as double precision floating-point numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-13.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Operation

```
// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD
```

VGETEXPSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
   THEN DEST[63:0] :=
            ConvertExpDPFP(SRC2[63:0])
   ELSE
       IF *merging-masking*                   ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                              ; zeroing-masking
                DEST[63:0] := 0
       FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPSD __m128d _mm_getexp_sd( __m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_maskz_getexp_sd( __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_getexp_round_sd( __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_maskz_getexp_round_sd( __mmask8 k, __m128d a, __m128d b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VGETEXPSH—Convert Exponents of Scalar FP16 Values to FP16 Values

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 43 /r VGETEXPSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Convert the exponent of FP16 values in the low word of the source operand to FP16 results representing unbiased integer exponents, and stores the results in the low word of the destination register subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction extracts the biased exponents from the normalized FP16 representation of the low word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to an unbiased negative integer value. The integer value of the unbiased exponent is converted to an FP16 value and written to the low word element of the destination operand (the first operand) as an FP16 number.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-14.

The formula is:

GETEXP(x) = floor($\log_2(|x|)$)

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTSH). Thus, the VGETEXPSH instruction does not require software to handle SIMD floating-point exceptions.

Operation

VGETEXPSH dest{k1}, src1, src2

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := getexp_fp16(src2.fp16[0]) // see VGETEXPPH
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPSH __m128h _mm_getexp_round_sh (__m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_mask_getexp_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_maskz_getexp_round_sh (__mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH __m128h _mm_getexp_sh (__m128h a, __m128h b);
VGETEXPSH __m128h _mm_mask_getexp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VGETEXPSH __m128h _mm_maskz_getexp_sh (__mmask8 k, __m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VGETEXPSS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	A	V/V	AVX512F OR AVX10.1	Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a single-precision floating-point value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Extracts the biased exponent from the normalized single precision floating-point representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single precision floating-point value and written to the destination operand (the first operand) as single precision floating-point numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

Operation

```
// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS
// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS
```

VGETEXPSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
            ConvertExpDPFP(SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0]:= 0
            FI
    FI;
ENDFOR
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETEXPSS __m128 _mm_getexp_ss( __m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss( __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss( __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss( __mmask8 k, __m128 a, __m128 b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VGETMANTPD—Extract Float64 Vector of Normalized Mantissas From Float64 Vector

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Convert double precision floating values in the source operand (the second operand) to double precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

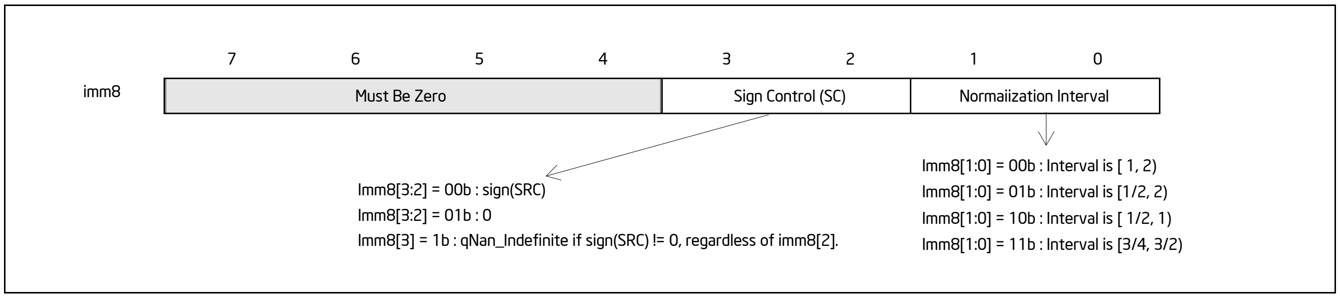

imm8
7
6
5
4
3
2
1
0
Must Be Zero
Sign Control (SC)
Normaiization Interval
Imm8[3:2] = 00b : sign(SRC)
Imm8[3:2] = 01b : 0
Imm8[3] = 1b : qNan_Indefinite if sign(SRC) != 0, regardless of imm8[2].
Imm8[1:0] = 00b : Interval is [1, 2)
Imm8[1:0] = 01b : Interval is [1/2, 2)
Imm8[1:0] = 10b : Interval is [1/2, 1)
Imm8[1:0] = 11b : Interval is [3/4, 3/2)

Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input double precision floating-point value x, The conversion operation is:

$$GetMant(x) = \pm 2^k|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted double precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b; otherwise instructions will #UD.

Table 5-16. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore *interv* If (SRC = SNaN) then #IE
+∞	1.0	Ignore *interv*
+0	1.0	Ignore *interv*

-0	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv*

-∞	IF (SC[1]) THEN {QNaN_Indefinite}
	ELSE {
	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv*
		If (SC[1]) then #IE

negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)[1]	If (SC[1]) then #IE

NOTES:

1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

Operation

```
def getmant_fp64(src, sign_control, normalization_interval):
    bias := 1023
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0x7FF) and (dst.fraction = 0)
    nan := (dst.exp = 0x7FF) and (dst.fraction != 0)
    src_signaling := src.fraction[51]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite

    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[51]
            dst.fraction := dst.fraction << 1
            dst.exp : = dst.exp - 1
        MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp := unbiased_exp[0]
    signaling_bit := dst.fraction[51]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
    return dst
```

VGETMANTPD (EVEX Encoded Versions)

```
VGETMANTPD dest{k1}, src, imm8
VL = 128, 256, or 512
KL := VL / 64
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.double[0]
        ELSE:
            tsrc := src.double[i]
        DEST.double[i] := getmant_fp64(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.double[i] := 0
    //else DEST.double[i] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTPD __m512d _mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);
```

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETMANTPH—Extract FP16 Vector of Normalized Mantissas from FP16 Vector

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 26 /r /ib VGETMANTPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Get normalized mantissa from FP16 vector xmm2/m128/m16bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.
EVEX.256.NP.0F3A.W0 26 /r /ib VGETMANTPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Get normalized mantissa from FP16 vector ymm2/m256/m16bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.
EVEX.512.NP.0F3A.W0 26 /r /ib VGETMANTPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Get normalized mantissa from FP16 vector zmm2/m512/m16bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction converts the FP16 values in the source operand (the second operand) to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-17. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

The destination elements are updated according to the writemask.

Table 5-17. imm8 Controls for VGETMANTPH/VGETMANTSH

imm8 Bits	Definition
imm8[7:4]	Must be zero.
imm8[3:2]	Sign Control (SC) 0b00: Sign(SRC) 0b01: 0 0b1x: QNaN_Indefinite if sign(SRC)!=0
imm8[1:0]	Interv 0b00: Interval is [1, 2) 0b01: Interval is [1/2, 2) 0b10: Interval is [1/2, 1) 0b11: Interval is [3/4, 3/2)

For each input FP16 value x, The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k|x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 5-17.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-18 when dealing with floating-point special numbers.

Table 5-18. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore *interv.* If (SRC = SNaN), then #IE.
+∞	1.0	Ignore *interv.*
+0	1.0	Ignore *interv.*

-0	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv.*

-∞	IF (SC[1]) THEN {QNaN_Indefinite}
	ELSE {
	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv.*
		If (SC[1]), then #IE.

negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)[1]	If (SC[1]), then #IE.

NOTES:

1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

Operation

```
def getmant_fp16(src, sign_control, normalization_interval):
    bias := 15
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and (dst.fraction = 0)
    denormal := (dst.exp = 0) and (dst.fraction != 0)
    infinity := (dst.exp = 0x1F) and (dst.fraction = 0)
    nan := (dst.exp = 0x1F) and (dst.fraction != 0)
    src_signaling := src.fraction[9]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite
    if denormal:
        jbit := 0
        dst.exp := bias            // set exponent to bias value
        while jbit = 0:
            jbit := dst.fraction[9]
            dst.fraction := dst.fraction << 1
            dst.exp : = dst.exp - 1
        MXCSR.DE := 1

    unbaiased_exp := dst.exp - bias
    odd_exp := unbaiased_exp[0]
    signaling_bit := dst.fraction[9]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
    return dst
```

VGETMANTPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := getmant_fp16(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPH __m128h _mm_getmant_ph (__m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_mask_getmant_ph (__m128h src, __mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_maskz_getmant_ph (__mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_getmant_ph (__m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_mask_getmant_ph (__m256h src, __mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_maskz_getmant_ph (__mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_mask_getmant_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_maskz_getmant_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_round_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_mask_getmant_round_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_maskz_getmant_round_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Convert single precision floating values in the source operand (the second operand) to single precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input single precision floating-point value x, The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted single precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

```
def getmant_fp32(src, sign_control, normalization_interval):
    bias := 127
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0xFF) and (dst.fraction = 0)
    nan := (dst.exp = 0xFF) and (dst.fraction != 0)
    src_signaling := src.fraction[22]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)

    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite

    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[22]
            dst.fraction := dst.fraction << 1
            dst.exp : = dst.exp - 1
        MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp  := unbiased_exp[0]
    signaling_bit := dst.fraction[22]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
    return dst
```

VGETMANTPS (EVEX encoded versions)

```
VGETMANTPS dest{k1}, src, imm8
VL = 128, 256, or 512
KL := VL / 32
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.float[0]
        ELSE:
            tsrc := src.float[i]
        DEST.float[i] := getmant_fp32(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.float[i] := 0
    //else DEST.float[i] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTPS __m512 _mm512_getmant_ps( __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn;
VGETMANTPS __m512 _mm512_maskz_getmant_ps(__mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_getmant_round_ps( __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m256 _mm256_getmant_ps( __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_maskz_getmant_ps( __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_getmant_ps( __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_maskz_getmant_ps( __mmask8 k, __m128 a, enum intv, enum sgn);
```

SIMD Floating-Point Exceptions

Denormal, Invalid.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VGETMANTSD—Extract Float64 of Normalized Mantissa From Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Extract the normalized mantissa of the low float64 element in xmm3/m64 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Convert the double precision floating values in the low quadword element of the second source operand (the third operand) to double precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted double precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Operation

```
// getmant_fp64(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPD
```

VGETMANTSD (EVEX encoded version)

```
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[63:0] :=
            getmant_fp64(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VGETMANTSH—Extract FP16 of Normalized Mantissa from FP16 Scalar

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 27 /r /ib VGETMANTSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Extract the normalized mantissa of the low FP16 element in xmm3/m16 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 subject to writemask k1 and merge with the other elements of xmm2. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction converts the FP16 value in the low element of the second source operand to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-17. The converted result is written to the low element of the destination operand using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For each input FP16 value x, The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k|x.\text{significand}|$$

where:

$$1 \le |x.\text{significand}| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 5-17.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-18 when dealing with floating-point special numbers.

Operation

```
VGETMANTSH dest{k1}, src1, src2, imm8
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

IF k1[0] or *no writemask*:
    dest.fp16[0] := getmant_fp16(src2.fp16[0],     // see VGETMANTPH
                                 sign_control,
                                 normalization_interval)
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else dest.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSH __m128h _mm_getmant_round_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm,
           _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTSH __m128h _mm_mask_getmant_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b,
           _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTSH __m128h _mm_maskz_getmant_round_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm,
           _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTSH __m128h _mm_getmant_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm,
           _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTSH __m128h _mm_mask_getmant_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM
           norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTSH __m128h _mm_maskz_getmant_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm,
           _MM_MANTISSA_SIGN_ENUM sign);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VGETMANTSS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Convert the single precision floating values in the low doubleword element of the second source operand (the third operand) to single precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$\text{GetMant}(x) = \pm 2^{k}|x.\text{significand}|$$

where:

$$1 <= |x.\text{significand}| < 2$$

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted single precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Operation

// getmant_fp32(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPS

VGETMANTSS (EVEX encoded version)

```
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
            getmant_fp32(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX	Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.256.66.0F3A.W0 18 /r ib VINSERTF32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W0 18 /r ib VINSERTF32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F OR AVX10.1	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.256.66.0F3A.W1 18 /r ib VINSERTF64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W1 18 /r ib VINSERTF64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ OR AVX10.1	Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W0 1A /r ib VINSERTF32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ OR AVX10.1	Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W1 1A /r ib VINSERTF64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F OR AVX10.1	Insert 256 bits of packed double precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.

Operation

VINSERTF32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:   TMP_DEST[127:0] := SRC2[127:0]
        1:   TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF64x2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:   TMP_DEST[127:0] := SRC2[127:0]
        01:   TMP_DEST[255:128] := SRC2[127:0]
        10:   TMP_DEST[383:256] := SRC2[127:0]
        11:   TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF32x8 (EVEX.U1.512 encoded version)

```
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF64x4 (EVEX.512 encoded version)

```
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTF128 (VEX encoded version)

```
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] := SRC2[127:0]
    1: TEMP[255:128] := SRC2[127:0]
ESAC
DEST := TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VINSERTF32x4 __m512 _mm512_insertf32x4( __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_maskz_insertf32x4( __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_insertf32x4( __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_maskz_insertf32x4( __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 _mm512_insertf32x8( __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_maskz_insertf32x8( __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d _mm512_insertf64x2( __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_maskz_insertf64x2( __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_insertf64x2( __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_mask_insertf64x2(__m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_maskz_insertf64x2( __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x4 __m512d _mm512_insertf64x4( __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_maskz_insertf64x4( __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);
VINSERTF128 __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);
VINSERTF128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-56, “Type E6NF Class Exception Conditions.”

VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

Opcode/ Instruction	Op / En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 38 /r ib VINSERTI128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX2	Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.256.66.0F3A.W0 38 /r ib VINSERTI32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W0 38 /r ib VINSERTI32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F OR AVX10.1	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.256.66.0F3A.W1 38 /r ib VINSERTI64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.512.66.0F3A.W1 38 /r ib VINSERTI64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ OR AVX10.1	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W0 3A /r ib VINSERTI32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ OR AVX10.1	Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.512.66.0F3A.W1 3A /r ib VINSERTI64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F OR AVX10.1	Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

VINSERTI32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:   TMP_DEST[127:0] := SRC2[127:0]
        1:   TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI64x2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] := SRC2[127:0]
        1:    TMP_DEST[255:128] := SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] := SRC2[127:0]
        01:  TMP_DEST[255:128] := SRC2[127:0]
        10:  TMP_DEST[383:256] := SRC2[127:0]
        11:  TMP_DEST[511:384] := SRC2[127:0]
    ESAC.
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI32x8 (EVEX.U1.512 encoded version)

```
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI64x4 (EVEX.512 encoded version)

```
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VINSERTI128

```
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] := SRC2[127:0]
    1: TEMP[255:128] := SRC2[127:0]
ESAC
DEST := TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VINSERTI32x4 _mm512i _inserti32x4( __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _mask_inserti32x4(__m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _maskz_inserti32x4( __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_inserti32x4( __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_mask_inserti32x4(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_maskz_inserti32x4( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 __m512i _mm512_inserti32x8( __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_mask_inserti32x8(__m512i s, __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_maskz_inserti32x8( __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI64x2 __m512i _mm512_inserti64x2( __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_mask_inserti64x2(__m512i s, __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_maskz_inserti64x2( __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_inserti64x2( __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_mask_inserti64x2(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_maskz_inserti64x2( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x4 _mm512_inserti64x4( __m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_mask_inserti64x4(__m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_maskz_inserti64x4( __mmask m, __m512i a, __m256i b, int imm);
VINSERTI128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-56, “Type E6NF Class Exception Conditions.”

VMAXPH—Return Maximum of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5F /r VMAXPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Return the maximum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5F /r VMAXPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Return the maximum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5F /r VMAXPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Return the maximum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

```
def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMAXPH dest, src1, src2

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.fp16[j] := MAX(SRC1.fp16[j], tsrc2)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXPH __m128h _mm_mask_max_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h _mm_maskz_max_ph (__mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h _mm_max_ph (__m128h a, __m128h b);
VMAXPH __m256h _mm256_mask_max_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h _mm256_maskz_max_ph (__mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h _mm256_max_ph (__m256h a, __m256h b);
VMAXPH __m512h _mm512_mask_max_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h _mm512_maskz_max_ph (__mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h _mm512_max_ph (__m512h a, __m512h b);
VMAXPH __m512h _mm512_mask_max_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h _mm512_maskz_max_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h _mm512_max_round_ph (__m512h a, __m512h b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VMAXSH—Return Maximum of Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5F /r VMAXSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Return the maximum low FP16 value between xmm3/m16 and xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the maximum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMAXSH dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MAX(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSH __m128h _mm_mask_max_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_maskz_max_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_max_round_sh (__m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_mask_max_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_maskz_max_sh (__mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_max_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VMINPH—Return Minimum of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5D /r VMINPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Return the minimum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5D /r VMINPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Return the minimum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5D /r VMINPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Return the minimum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

```
def MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

```
VMINPH dest, src1, src2
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            tsrc2 := SRC2.fp16[0]
        ELSE:
            tsrc2 := SRC2.fp16[j]
        DEST.fp16[j] := MIN(SRC1.fp16[j], tsrc2)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPH __m128h _mm_mask_min_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_maskz_min_ph (__mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_min_ph (__m128h a, __m128h b);
VMINPH __m256h _mm256_mask_min_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_maskz_min_ph (__mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_min_ph (__m256h a, __m256h b);
VMINPH __m512h _mm512_mask_min_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_maskz_min_ph (__mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_min_ph (__m512h a, __m512h b);
VMINPH __m512h _mm512_mask_min_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_maskz_min_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_min_round_ph (__m512h a, __m512h b, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VMINSH—Return Minimum Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5D /r VMINSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Return the minimum low FP16 value between xmm3/m16 and xmm2. Stores the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the minimum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
def MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

VMINSH dest, src1, src2

```
IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MIN(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSH __m128h _mm_mask_min_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h _mm_maskz_min_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h _mm_min_round_sh (__m128h a, __m128h b, int sae);
VMINSH __m128h _mm_mask_min_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINSH __m128h _mm_maskz_min_sh (__mmask8 k, __m128h a, __m128h b);
VMINSH __m128h _mm_min_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VMOVSH—Move Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 10 /r VMOVSH xmm1{k1}{z}, m16	A	V/V	AVX512-FP16 OR AVX10.1	Move FP16 value from m16 to xmm1 subject to writemask k1.
EVEX.LLIG.F3.MAP5.W0 11 /r VMOVSH m16{k1}, xmm1	B	V/V	AVX512-FP16 OR AVX10.1	Move low FP16 value from xmm1 to m16 subject to writemask k1.
EVEX.LLIG.F3.MAP5.W0 10 /r VMOVSH xmm1{k1}{z}, xmm2, xmm3	C	V/V	AVX512-FP16 OR AVX10.1	Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].
EVEX.LLIG.F3.MAP5.W0 11 /r VMOVSH xmm1{k1}{z}, xmm2, xmm3	D	V/V	AVX512-FP16 OR AVX10.1	Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
C	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
D	N/A	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	N/A

Description

This instruction moves a FP16 value to a register or memory location.

The two register-only forms are aliases and differ only in where their operands are encoded; this is a side effect of the encodings selected.

Operation

VMOVSH dest, src (two operand load)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[MAXVL:16] := 0
```

VMOVSH dest, src (two operand store)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC.fp16[0]
// ELSE DEST.fp16[0] remains unchanged
```

VMOVSH dest, src1, src2 (three operand copy)

```
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSH __m128h _mm_load_sh (void const* mem_addr);
VMOVSH __m128h _mm_mask_load_sh (__m128h src, __mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_maskz_load_sh (__mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_mask_move_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_maskz_move_sh (__mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_move_sh (__m128h a, __m128h b);
VMOVSH void _mm_mask_store_sh (void * mem_addr, __mmask8 k, __m128h a);
VMOVSH void _mm_store_sh (void * mem_addr, __m128h a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E5 Class Exception Conditions.”

VMOVW—Move Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP5.WIG 6E /r VMOVW xmm1, reg/m16	A	V/V	AVX512-FP16 OR AVX10.1	Copy word from reg/m16 to xmm1.
EVEX.128.66.MAP5.WIG 7E /r VMOVW reg/m16, xmm1	B	V/V	AVX512-FP16 OR AVX10.1	Copy word from xmm1 to reg/m16.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

This instruction either (a) copies one word element from an XMM register to a general-purpose register or memory location or (b) copies one word element from a general-purpose register or memory location to an XMM register. When writing a general-purpose register, the lower 16-bits of the register will contain the word value. The upper bits of the general-purpose register are written with zeros.

Operation

VMOVW dest, src (two operand load)

```
DEST.word[0] := SRC.word[0]
DEST[MAXVL:16] := 0
```

VMOVW dest, src (two operand store)

```
DEST.word[0] := SRC.word[0]
// upper bits of GPR DEST are zeroed
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVW short _mm_cvtsi128_si16 (__m128i a);
VMOVW __m128i _mm_cvtsi16_si128 (short a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Table 2-59, “Type E9NF Class Exception Conditions.”

VMULPH—Multiply Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 59 /r VMULPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from xmm3/m128/m16bcst to xmm2 and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 59 /r VMULPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Multiply packed FP16 values from ymm3/m256/m16bcst to ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 59 /r VMULPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply packed FP16 values in zmm3/m512/m16bcst with zmm2 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction multiplies packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VMULPH (EVEX encoded versions) when src2 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VMULPH (EVEX encoded versions) when src2 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULPH __m128h _mm_mask_mul_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_maskz_mul_ph (__mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_mul_ph (__m128h a, __m128h b);
VMULPH __m256h _mm256_mask_mul_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_maskz_mul_ph (__mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_mul_ph (__m256h a, __m256h b);
VMULPH __m512h _mm512_mask_mul_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_maskz_mul_ph (__mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_mul_ph (__m512h a, __m512h b);
VMULPH __m512h _mm512_mask_mul_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_maskz_mul_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_mul_round_ph (__m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E2 Class Exception Conditions.”

VMULSH—Multiply Scalar FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 59 /r VMULSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Multiply the low FP16 value in xmm3/m16 by low FP16 value in xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction multiplies the low FP16 value from the source operands and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VMULSH (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] * SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSH __m128h _mm_mask_mul_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_maskz_mul_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mul_round_sh (__m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mask_mul_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMULSH __m128h _mm_maskz_mul_sh (__mmask8 k, __m128h a, __m128h b);
VMULSH __m128h _mm_mul_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W0 66 /r VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W0 66 /r VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.
EVEX.128.66.0F38.W1 66 /r VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W1 66 /r VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W1 66 /r VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).

Operation

VPBLENDMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC2[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+7:i] := SRC1[i+7:i]
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

VPBLENDMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC2[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+15:i] := SRC1[i+15:i]
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBLENDMB __m512i _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i _mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i _mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i _mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W0 64 /r VPBLENDMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.
EVEX.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

Operation

VPBLENDMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

VPBLENDMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPBROADCAST—Load Integer and Broadcast

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in xmm1.
VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in ymm1.
EVEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1{k1}{z}, xmm2/m8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1{k1}{z}, xmm2/m8	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1{k1}{z}, xmm2/m8	B	V/V	AVX512BW OR AVX10.1	Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in xmm1.
VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in ymm1.
EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1{k1}{z}, xmm2/m16	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1{k1}{z}, xmm2/m16	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1{k1}{z}, xmm2/m16	B	V/V	AVX512BW OR AVX10.1	Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in ymm1.
EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1 {k1}{z}, xmm2/m32	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F OR AVX10.1	Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to two locations in xmm1.
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to four locations in ymm1.
EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F OR AVX10.1	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 59 /r VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 59 /r VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ OR AVX10.1	Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 ymm1, m128	A	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.
EVEX.256.66.0F38.W0 5A /r VBROADCASTI32X4 ymm1 {k1}{z}, m128	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 5A /r VBROADCASTI32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F OR AVX10.1	Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 5A /r VBROADCASTI64X2 ymm1 {k1}{z}, m128	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 5A /r VBROADCASTI64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ OR AVX10.1	Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 5B /r VBROADCASTI32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ OR AVX10.1	Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 5B /r VBROADCASTI64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F OR AVX10.1	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VBROADCASTI32X4 and VBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

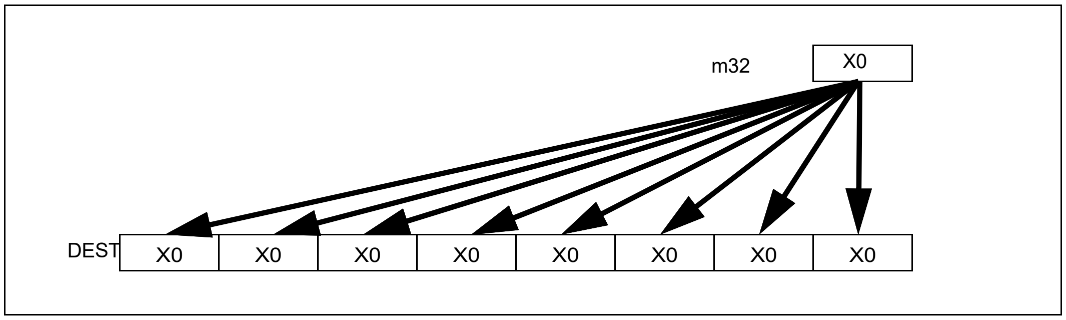


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

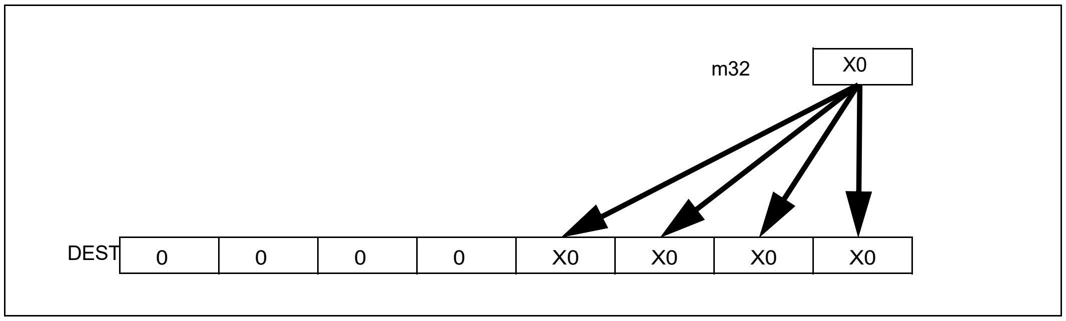


Figure 5-17. VPBROADCASTD Operation (128-bit version)

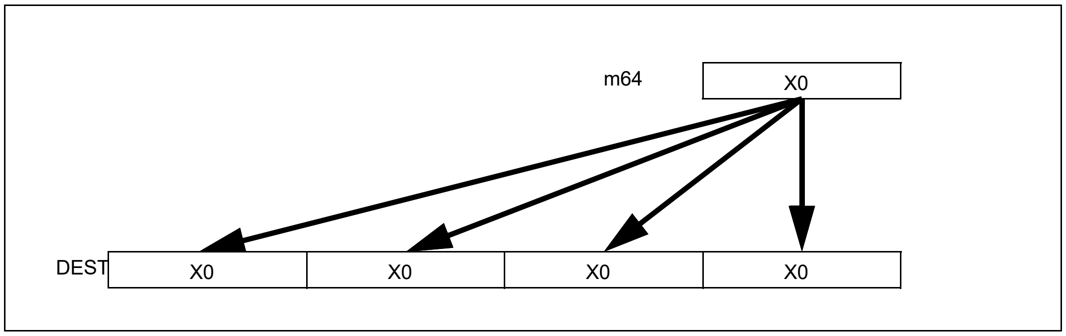


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

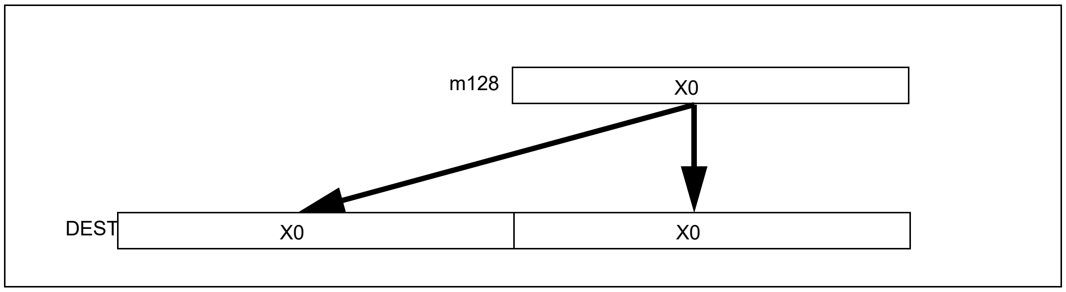


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

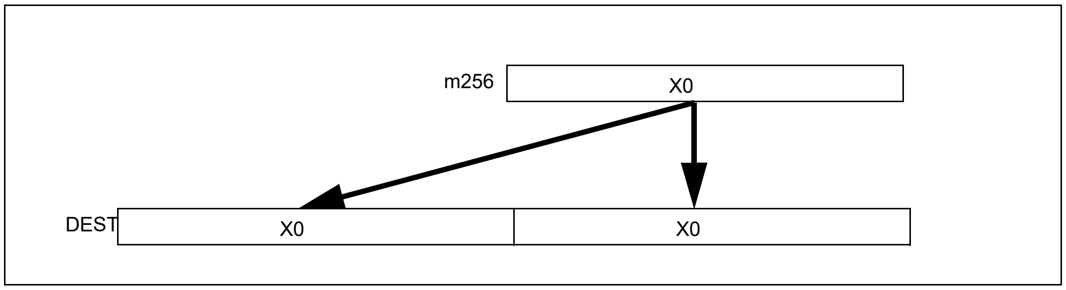


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[7:0]
        ELSE
            IF *merging-masking*                  ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[15:0]
        ELSE
            IF *merging-masking*                  ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTD (128 bit version)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[MAXVL-1:128] := 0
```

VPBROADCASTD (VEX.256 encoded version)

```
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[159:128] := temp
DEST[191:160] := temp
DEST[223:192] := temp
DEST[255:224] := temp
DEST[MAXVL-1:256] := 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTQ (VEX.256 encoded version)

```
temp := SRC[63:0]
DEST[63:0] := temp
DEST[127:64] := temp
DEST[191:128] := temp
DEST[255:192] := temp
DEST[MAXVL-1:256] := 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VBROADCASTI32x2 (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    n := (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] := 0
```

VBROADCASTI128 (VEX.256 encoded version)

```
temp := SRC[127:0]
DEST[127:0] := temp
DEST[255:128] := temp
DEST[MAXVL-1:256] := 0
```

VBROADCASTI32X4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j* 32
   n := (j modulo 4) * 32
   IF k1[j] OR *no writemask*
       THEN DEST[i+31:i] := SRC[n+31:n]
       ELSE
           IF *merging-masking*                ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE                            ; zeroing-masking
                   DEST[i+31:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTI64X2 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 64
   n := (j modulo 2) * 64
   IF k1[j] OR *no writemask*
       THEN DEST[i+63:i] := SRC[n+63:n]
       ELSE
           IF *merging-masking*                ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
               ELSE                            ; zeroing-masking
                   DEST[i+63:i] = 0
           FI
   FI;
ENDFOR;
```

VBROADCASTI32X8 (EVEX.U1.512 encoded version)

```
FOR j := 0 TO 15
   i := j * 32
   n := (j modulo 8) * 32
   IF k1[j] OR *no writemask*
       THEN DEST[i+31:i] := SRC[n+31:n]
       ELSE
           IF *merging-masking*                ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE                            ; zeroing-masking
                   DEST[i+31:i] := 0
           FI
   FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VBROADCASTI64X4 (EVEX.512 encoded version)

```
FOR j := 0 TO 7
    i := j * 64
    n := (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBROADCASTB __m512i _mm512_broadcastb_epi8( __m128i a);
VPBROADCASTB __m512i _mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i _mm512_maskz_broadcastb_epi8( __mmask64 k, __m128i a);
VPBROADCASTB __m256i _mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i _mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i _mm256_maskz_broadcastb_epi8( __mmask32 k, __m128i a);
VPBROADCASTB __m128i _mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_maskz_broadcastb_epi8( __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_broadcastb_epi8(__m128i a);
VPBROADCASTD __m512i _mm512_broadcastd_epi32( __m128i a);
VPBROADCASTD __m512i _mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i _mm512_maskz_broadcastd_epi32( __mmask16 k, __m128i a);
VPBROADCASTD __m256i _mm256_broadcastd_epi32( __m128i a);
VPBROADCASTD __m256i _mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i _mm256_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_broadcastd_epi32(__m128i a);
VPBROADCASTD __m128i _mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m512i _mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i _mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTW __m512i _mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i _mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i _mm512_maskz_broadcastw_epi16( __mmask32 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i _mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i _mm256_maskz_broadcastw_epi16( __mmask16 k, __m128i a);
VPBROADCASTW __m128i _mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i _mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i _mm_maskz_broadcastw_epi16( __mmask8 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_broadcast_i32x2( __m128i a);
VBROADCASTI32x2 __m512i _mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_maskz_broadcast_i32x2( __mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_broadcast_i32x2( __m128i a);
VBROADCASTI32x2 __m256i _mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_maskz_broadcast_i32x2( __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m128i _mm_mask_broadcast_i32x2(__m128i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_maskz_broadcast_i32x2( __mmask8 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_broadcast_i32x4( __m128i a);
VBROADCASTI32x4 __m512i _mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_maskz_broadcast_i32x4( __mmask16 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_broadcast_i32x4( __m128i a);
VBROADCASTI32x4 __m256i _mm256_mask_broadcast_i32x4(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_maskz_broadcast_i32x4( __mmask8 k, __m128i a);
VBROADCASTI32x8 __m512i _mm512_broadcast_i32x8( __m256i a);
VBROADCASTI32x8 __m512i _mm512_mask_broadcast_i32x8(__m512i s, __mmask16 k, __m256i a);
VBROADCASTI32x8 __m512i _mm512_maskz_broadcast_i32x8( __mmask16 k, __m256i a);
VBROADCASTI64x2 __m512i _mm512_broadcast_i64x2( __m128i a);
VBROADCASTI64x2 __m512i _mm512_mask_broadcast_i64x2(__m512i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m512i _mm512_maskz_broadcast_i64x2( __mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_broadcast_i64x2( __m128i a);
VBROADCASTI64x2 __m256i _mm256_mask_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_maskz_broadcast_i64x2( __mmask8 k, __m128i a);
VBROADCASTI64x4 __m512i _mm512_broadcast_i64x4( __m256i a);
VBROADCASTI64x4 __m512i _mm512_mask_broadcast_i64x4(__m512i s, __mmask8 k, __m256i a);
VBROADCASTI64x4 __m512i _mm512_maskz_broadcast_i64x4( __mmask8 k, __m256i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."

EVEX-encoded instructions, syntax with reg/mem operand, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
	If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.
	If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

VPBROADCASTB/W/D/Q—Load With Broadcast Integer Data From General Purpose Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 {k1}{z}, reg	A	V/V	AVX512BW OR AVX10.1	Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 {k1}{z}, reg	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 {k1}{z}, reg	A	V/V	AVX512BW OR AVX10.1	Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 {k1}{z}, r32	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 32-bit value from a GPR to all doublewords in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 32-bit value from a GPR to all doublewords in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	A	V/V	AVX512F OR AVX10.1	Broadcast a 32-bit value from a GPR to all doublewords in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64	A	V/N.E.[1]	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 64-bit value from a GPR to all quadwords in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64	A	V/N.E.[1]	(AVX512VL AND AVX512F) OR AVX10.1	Broadcast a 64-bit value from a GPR to all quadwords in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	A	V/N.E.[1]	AVX512F OR AVX10.1	Broadcast a 64-bit value from a GPR to all quadwords in the 512-bit destination subject to writemask k1.

NOTES:

1. EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SRC[7:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBROADCASTB __m512i _mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i _mm512_maskz_set1_epi8( __mmask64 k, int a);
VPBROADCASTB __m256i _mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i _mm256_maskz_set1_epi8( __mmask32 k, int a);
VPBROADCASTB __m128i _mm_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i _mm_maskz_set1_epi8( __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_mask_set1_epi32(__m512i s, __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_maskz_set1_epi32( __mmask16 k, int a);
VPBROADCASTD __m256i _mm256_mask_set1_epi32(__m256i s, __mmask8 k, int a);
VPBROADCASTD __m256i _mm256_maskz_set1_epi32( __mmask8 k, int a);
VPBROADCASTD __m128i _mm_mask_set1_epi32(__m128i s, __mmask8 k, int a);
VPBROADCASTD __m128i _mm_maskz_set1_epi32( __mmask8 k, int a);
VPBROADCASTQ __m512i _mm512_mask_set1_epi64(__m512i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m512i _mm512_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_mask_set1_epi64(__m256i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_mask_set1_epi64(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTW __m512i _mm512_mask_set1_epi16(__m512i s, __mmask32 k, int a);
VPBROADCASTW __m512i _mm512_maskz_set1_epi16( __mmask32 k, int a);
VPBROADCASTW __m256i _mm256_mask_set1_epi16(__m256i s, __mmask16 k, int a);
VPBROADCASTW __m256i _mm256_maskz_set1_epi16( __mmask16 k, int a);
VPBROADCASTW __m128i _mm_mask_set1_epi16(__m128i s, __mmask8 k, int a);
VPBROADCASTW __m128i _mm_maskz_set1_epi16( __mmask8 k, int a);
```

Exceptions

EVEX-encoded instructions, see Table 2-57, "Type E7NM Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPBROADCASTM—Broadcast Mask to Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low byte value in k1 to two locations in xmm1.
EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low byte value in k1 to four locations in ymm1.
EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1	RM	V/V	AVX512CD OR AVX10.1	Broadcast low byte value in k1 to eight locations in zmm1.
EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low word value in k1 to four locations in xmm1.
EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1	RM	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Broadcast low word value in k1 to eight locations in ymm1.
EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1	RM	V/V	AVX512CD OR AVX10.1	Broadcast low word value in k1 to sixteen locations in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTMB2Q

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    DEST[i+63:i] := ZeroExtend(SRC[7:0])
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPBROADCASTMW2D

```
(KL, VL) = (4, 128), (8, 256),(16, 512)
FOR j := 0 TO KL-1
    i := j*32
    DEST[i+31:i] := ZeroExtend(SRC[15:0])
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPBROADCASTMB2Q __m512i _mm512_broadcastmb_epi64( __mmask8);
VPBROADCASTMW2D __m512i _mm512_broadcastmw_epi32( __mmask16);
VPBROADCASTMB2Q __m256i _mm256_broadcastmb_epi64( __mmask8);
VPBROADCASTMW2D __m256i _mm256_broadcastmw_epi32( __mmask8);
VPBROADCASTMB2Q __m128i _mm_broadcastmb_epi64( __mmask8);
VPBROADCASTMW2D __m128i _mm_broadcastmw_epi32( __mmask8);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-56, “Type E6NF Class Exception Conditions.”

VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.128.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Table 5-19. Pseudo-Op and VPCMP* Implementation

Pseudo-Op	**PCMPM Implementation**
VPCMPEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 0*
VPCMPLT* *reg1, reg2, reg3*	VPCMP**reg1, reg2, reg3, 1*
VPCMPLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 2*
VPCMPNEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 4*
VPPCMPNLT* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 5*
VPCMPNLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 6*

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
         THEN
              CMP := SRC1[i+7:i] OP SRC2[i+7:i];
              IF CMP = TRUE
                   THEN DEST[j] := 1;
                   ELSE DEST[j] := 0; FI;
         ELSE      DEST[j] = 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCMPB __mmask64 _mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __mmask64 _mm512_mask_cmp_epi8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask32 _mm256_mask_cmp_epi8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 _mm_cmp_epi8_mask( __m128i a, __m128i b, int cmp);
VPCMPB __mmask16 _mm_mask_cmp_epi8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m512i a, __m512i b);
VPCMPB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m256i a, __m256i b);
VPCMPB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m128i a, __m128i b);
VPCMPB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask16 m, __m128i a, __m128i b);
VPCMPUB __mmask64 _mm512_cmp_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPUB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.
EVEX.128.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPD __mmask16 _mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_epu32_mask(__m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_mask_cmp_epu32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 _mm256_cmp_epi32_mask(__m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m256i a, __m256i b);
VPCMPD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_cmp_epu32_mask(__m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_mask_cmp_epu32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 _mm_cmp_epi32_mask(__m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m128i a, __m128i b);
VPCMPD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp_epu32_mask(__m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_mask_cmp_epu32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m128i a, __m128i b);
VPCMPUD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.128.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0            ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] := 0            ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPQ __mmask8 _mm512_cmp_epi64_mask(__m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_cmp_epu64_mask(__m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 _mm256_cmp_epi64_mask(__m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m256i a, __m256i b);
VPCMPQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_cmp_epu64_mask(__m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_mask_cmp_epu64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8 _mm_cmp_epi64_mask(__m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_cmp_epu64_mask(__m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_mask_cmp_epu64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-51, "Type E4 Class Exception Conditions."

VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.128.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW OR AVX10.1	Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-19.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP := EQ;
    1: OP := LT;
    2: OP := LE;
    3: OP := FALSE;
    4: OP := NEQ;
    5: OP := NLT;
    6: OP := NLE;
    7: OP := TRUE;
ESAC;
```

VPCMPW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            ICMP := SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPCMPUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE    DEST[j] = 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCMPW __mmask32 _mm512_cmp_epi16_mask( __m512i a, __m512i b, int cmp);
VPCMPW __mmask32 _mm512_mask_cmp_epi16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 _mm256_cmp_epi16_mask( __m256i a, __m256i b, int cmp);
VPCMPW __mmask16 _mm256_mask_cmp_epi16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 _mm_cmp_epi16_mask( __m128i a, __m128i b, int cmp);
VPCMPW __mmask8 _mm_mask_cmp_epi16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m512i a, __m512i b);
VPCMPW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m256i a, __m256i b);
VPCMPW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m128i a, __m128i b);
VPCMPW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 _mm512_cmp_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp_epu16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp_epu16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp_epu16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp_epu16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPCOMPRESSB/VCOMPRESSW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB m128{k1}, xmm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed byte values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB m256{k1}, ymm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed byte values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB m512{k1}, zmm1	A	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed byte values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW m128{k1}, xmm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed word values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW m256{k1}, ymm1	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed word values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Compress up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW m512{k1}, zmm1	A	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed word values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Compress up to 512 bits of packed word values from zmm2 to zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A
B	N/A	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 64 byte values or 32 word values from the source operand (second operand) to the destination operand (first operand), based on the active elements determined by the writemask operand. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves up to 512 bits of packed byte values from the source operand (second operand) to the destination operand (first operand). This instruction is used to store partial contents of a vector register into a byte vector or single memory location using the active elements in operand writemask.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSB store form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] := SRC.byte[j]
        k := k +1
```

VPCOMPRESSB reg-reg form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] := SRC.byte[j]
        k := k + 1
IF *merging-masking*:
    *DEST[VL-1:k*8] remains unchanged*
    ELSE DEST[VL-1:k*8] := 0
DEST[MAX_VL-1:VL] := 0
```

VPCOMPRESSW store form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] := SRC.word[j]
        k := k + 1
```

VPCOMPRESSW reg-reg form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] := SRC.word[j]
        k := k + 1
IF *merging-masking*:
    *DEST[VL-1:k*16] remains unchanged*
    ELSE DEST[VL-1:k*16] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCOMPRESSB __m128i _mm_mask_compress_epi8(__m128i, __mmask16, __m128i);
VPCOMPRESSB __m128i _mm_maskz_compress_epi8(__mmask16, __m128i);
VPCOMPRESSB __m256i _mm256_mask_compress_epi8(__m256i, __mmask32, __m256i);
VPCOMPRESSB __m256i _mm256_maskz_compress_epi8(__mmask32, __m256i);
VPCOMPRESSB __m512i _mm512_mask_compress_epi8(__m512i, __mmask64, __m512i);
VPCOMPRESSB __m512i _mm512_maskz_compress_epi8(__mmask64, __m512i);
VPCOMPRESSB  void _mm_mask_compressstoreu_epi8(void*, __mmask16, __m128i);
VPCOMPRESSB  void _mm256_mask_compressstoreu_epi8(void*, __mmask32, __m256i);
VPCOMPRESSB  void _mm512_mask_compressstoreu_epi8(void*, __mmask64, __m512i);
VPCOMPRESSW  __m128i  _mm_mask_compress_epi16(__m128i, __mmask8, __m128i);
VPCOMPRESSW  __m128i  _mm_maskz_compress_epi16(__mmask8, __m128i);
VPCOMPRESSW  __m256i  _mm256_mask_compress_epi16(__m256i, __mmask16, __m256i);
VPCOMPRESSW  __m256i  _mm256_maskz_compress_epi16(__mmask16, __m256i);
VPCOMPRESSW  __m512i  _mm512_mask_compress_epi16(__m512i, __mmask32, __m512i);
VPCOMPRESSW  __m512i  _mm512_maskz_compress_epi16(__mmask32, __m512i);
VPCOMPRESSW  void  _mm_mask_compressstoreu_epi16(void*, __mmask8, __m128i);
VPCOMPRESSW  void  _mm256_mask_compressstoreu_epi16(void*, __mmask16, __m256i);
VPCOMPRESSW  void  _mm512_mask_compressstoreu_epi16(void*, __mmask32, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed doubleword integer values from xmm2 to xmm1/m128 using control mask k1.
EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed doubleword integer values from ymm2 to ymm1/m256 using control mask k1.
EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed doubleword integer values from zmm2 to zmm1/m512 using control mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSD (EVEX encoded versions) store form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR;
```

VPCOMPRESSD (EVEX encoded versions) reg-reg form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32( __mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32( __mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32( __mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8B /r VPCOMPRESSQ xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed quadword integer values from xmm2 to xmm1/m128 using control mask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Compress packed quadword integer values from ymm2 to ymm1/m256 using control mask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Compress packed quadword integer values from zmm2 to zmm1/m512 using control mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSQ (EVEX encoded versions) store form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENFOR
```

VPCOMPRESSQ (EVEX encoded versions) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCOMPRESSQ __m512i _mm512_mask_compress_epi64(__m512i s, __mmask8 c, __m512i a);
VPCOMPRESSQ __m512i _mm512_maskz_compress_epi64( __mmask8 c, __m512i a);
VPCOMPRESSQ void _mm512_mask_compressstoreu_epi64(void * a, __mmask8 c, __m512i s);
VPCOMPRESSQ __m256i _mm256_mask_compress_epi64(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSQ __m256i _mm256_maskz_compress_epi64( __mmask8 c, __m256i a);
VPCOMPRESSQ void _mm256_mask_compressstoreu_epi64(void * a, __mmask8 c, __m256i s);
VPCOMPRESSQ __m128i _mm_mask_compress_epi64(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSQ __m128i _mm_maskz_compress_epi64( __mmask8 c, __m128i a);
VPCOMPRESSQ void _mm_mask_compressstoreu_epi64(void * a, __mmask8 c, __m128i s);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values Into Dense Memory/ Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD OR AVX10.1	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD OR AVX10.1	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element’s comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPCONFLICTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask* THEN
        FOR k := 0 TO j-1
            m := k*32
            IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
                DEST[i+k] := 1
            ELSE
                DEST[i+k] := 0
            FI
        ENDFOR
        DEST[i+31:i+j] := 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+31:i] remains unchanged*
        ELSE
            DEST[i+31:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPCONFLICTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask* THEN
        FOR k := 0 TO j-1
            m := k*64
            IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
                DEST[i+k] := 1
            ELSE
                DEST[i+k] := 0
            FI
        ENDFOR
        DEST[i+63:i+j] := 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCONFLICTD __m512i _mm512_conflict_epi32( __m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64( __m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32( __m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64( __m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32( __m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64( __m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

VPDPBUSD—Multiply and Add Unsigned and Signed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.
EVEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs of signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs of signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 50 /r VPDPBUSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 4 pairs of signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand.

This instruction supports memory fault suppression.

Operation

VPDPBUSD dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:

    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
    DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAX_VL-1:VL] := 0
```

VPDPBUSD dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
        DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPBUSD __m128i _mm_dpbusd_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_mask_dpbusd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSD __m128i _mm_maskz_dpbusd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSD __m256i _mm256_dpbusd_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPDPBUSDS—Multiply and Add Unsigned and Signed Bytes With Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 4 pairs signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.
VEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 4 pairs signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.
EVEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 4 pairs signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 51 /r VPDPBUSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 4 pairs signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result, with signed saturation in zmm1, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand. If the intermediate sum overflows a 32b signed number the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPBUSDS dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:
    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)

DEST[MAX_VL-1:VL] := 0
```

VPDPBUSDS dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) *SIGN_EXTEND(t.byte[3])
        DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:   // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPBUSDS __m128i _mm_dpbusds_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_mask_dpbusds_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSDS __m128i _mm_maskz_dpbusds_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSDS __m256i _mm256_dpbusds_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_dpbusds_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_mask_dpbusds_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_maskz_dpbusds_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSDS __m512i _mm512_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_mask_dpbusds_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_maskz_dpbusds_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPDPWSSD—Multiply and Add Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 2 pairs signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 2 pairs signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1.
EVEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 52 /r VPDPWSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

Operation

VPDPWSSD dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0] )
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1] )
    DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
DEST[MAX_VL-1:VL] := 0
```

VPDPWSSD dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
        p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
        DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPWSSD __m128i _mm_dpwssd_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSD __m256i _mm256_dpwssd_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPDPWSSDS—Multiply and Add Signed Word Integers With Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI	Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation.
VEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI	Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation.
EVEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.
EVEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VNNI AND AVX512VL) OR AVX10.1	Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.
EVEX.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VNNI OR AVX10.1	Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPWSSDS dest, src1, src2 (VEX encoded versions)

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0])
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
DEST[MAX_VL-1:VL] := 0
```

VPDPWSSDS dest, src1, src2 (EVEX encoded versions)

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
        p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
        DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE:     // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPDPWSSDS __m128i _mm_dpwssds_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSDS __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSDS __m256i _mm256_dpwssds_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPERMB—Permute Packed Bytes Elements

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8D /r VPERMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 8D /r VPERMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512_VBMI) OR AVX10.1	Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 8D /r VPERMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI OR AVX10.1	Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

Operation

VPERMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    n := 3;
ELSE IF VL = 256:
    n := 4;
ELSE IF VL = 512:
    n := 5;
FI;
FOR j := 0 TO KL-1:
    id := SRC1[j*8 + n : j*8] ; // location of the source byte
    IF k1[j] OR *no writemask* THEN
        DEST[j*8 + 7: j*8] := SRC2[id*8 +7: id*8];
    ELSE IF zeroing-masking THEN
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMB __m512i _mm512_permutexvar_epi8( __m512i idx, __m512i a);
VPERMB __m512i _mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);
VPERMB __m512i _mm512_maskz_permutexvar_epi8( __mmask64 k, __m512i idx, __m512i a);
VPERMB __m256i _mm256_permutexvar_epi8( __m256i idx, __m256i a);
VPERMB __m256i _mm256_mask_permutexvar_epi8(__m256i s, __mmask32 k, __m256i idx, __m256i a);
VPERMB __m256i _mm256_maskz_permutexvar_epi8( __mmask32 k, __m256i idx, __m256i a);
VPERMB __m128i _mm_permutexvar_epi8( __m128i idx, __m128i a);
VPERMB __m128i _mm_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i _mm_maskz_permutexvar_epi8( __mmask16 k, __m128i idx, __m128i a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VPERMD/VPERMW—Permute Packed Doubleword/Word Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.256.66.0F38.W0 36 /r VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 36 /r VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 8D /r VPERMW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 8D /r VPERMW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 8D /r VPERMW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW OR AVX10.1	Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

Operation

VPERMD (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n := 2; FI;
IF VL = 512 THEN n := 3; FI;
FOR j := 0 TO KL-1
    i := j * 32
    id := 32*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC2[31:0];
                ELSE DEST[i+31:i] := SRC2[id+31:id];
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMD (VEX.256 encoded version)

```
DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] := (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] := (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

VPERMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128 THEN n := 2; FI;
IF VL = 256 THEN n := 3; FI;
IF VL = 512 THEN n := 4; FI;
FOR j := 0 TO KL-1
    i := j * 16
    id := 16*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SRC2[id+15:id]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMD __m512i _mm512_permutexvar_epi32( __m512i idx, __m512i a);
VPERMD __m512i _mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i _mm512_maskz_permutexvar_epi32( __mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i _mm256_permutexvar_epi32( __m256i idx, __m256i a);
VPERMD __m256i _mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i _mm256_maskz_permutexvar_epi32( __mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i _mm512_permutexvar_epi16( __m512i idx, __m512i a);
VPERMW __m512i _mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i _mm512_maskz_permutexvar_epi16( __mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i _mm256_permutexvar_epi16( __m256i idx, __m256i a);
VPERMW __m256i _mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i _mm256_maskz_permutexvar_epi16( __mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i _mm_permutexvar_epi16( __m128i idx, __m128i a);
VPERMW __m128i _mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i _mm_maskz_permutexvar_epi16( __mmask8 k, __m128i idx, __m128i a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPERMD, see Table 2-52, “Type E4NF Class Exception Conditions.”

EVEX-encoded VPERMW, see Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.
	If EVEX.L’L = 0 for VPERMD.

VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 75 /r VPERMI2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 75 /r VPERMI2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AVX512_VBMI) OR AVX10.1	Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 75 /r VPERMI2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI OR AVX10.1	Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes byte values in the second operand (the first source operand) and the third operand (the second source operand) using the byte indices in the first operand (the destination operand) to select byte elements from the second or third source operands. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMI2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    id := 3;
ELSE IF VL = 256:
    id := 4;
ELSE IF VL = 512:
    id := 5;
FI;
TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
    off := 8*SRC1[j*8 + id: j*8] ;
    IF k1[j] OR *no writemask*:
        DEST[j*8 + 7: j*8] := TMP_DEST[j*8+id+1]? SRC2[off+7:off] : SRC1[off+7:off];
    ELSE IF *zeroing-masking*
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMI2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMI2B __m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i idx, __mmask64 k, __m512i b);
VPERMI2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMI2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMI2B __m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i idx, __mmask32 k, __m256i b);
VPERMI2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMI2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMI2B __m128i _mm_mask2_permutex2var_epi8(__m128i a, __m128i idx, __mmask16 k, __m128i b);
VPERMI2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 75 /r VPERMI2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 75 /r VPERMI2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 75 /r VPERMI2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 76 /r VPERMI2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 76 /r VPERMI2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 76 /r VPERMI2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 76 /r VPERMI2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 76 /r VPERMI2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 76 /r VPERMI2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 77 /r VPERMI2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 77 /r VPERMI2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 77 /r VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMI2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id := 2
FI;
IF VL = 256
    id := 3
FI;
IF VL = 512
    id := 4
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 16
    off := 16*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i]=TMP_DEST[i+id+1] ? SRC2[off+15:off]
                        : SRC1[off+15:off]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMI2D/VPERMI2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id := 1
FI;
IF VL = 256
    id := 2
FI;
IF VL = 512
    id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 32
    off := 32*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i]  := TMP_DEST[i+id+1] ? SRC2[31:0]
                        : SRC1[off+31:off]
            ELSE
                DEST[i+31:i] := TMP_DEST[i+id+1] ? SRC2[off+31:off]
                        : SRC1[off+31:off]
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMI2Q/VPERMI2PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id := 0
FI;
IF VL = 256
    id := 1
FI;
IF VL = 512
    id := 2
FI;
TMP_DEST:= DEST
FOR j := 0 TO KL-1
    i := j * 64
    off := 64*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := TMP_DEST[i+id+1] ? SRC2[63:0]
                        : SRC1[off+63:off]
            ELSE
                DEST[i+63:i] := TMP_DEST[i+id+1] ? SRC2[off+63:off]
                        : SRC1[off+63:off]
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMI __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMI2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMI2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMI2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMI2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMI2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMI2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMI2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMI2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMI2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMI2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMI2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMI2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMI2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);

```
VPERMI2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMI2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMI2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMI2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMI2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMI2D/Q/PS/PD: See Table 2-52, "Type E4NF Class Exception Conditions."

VPERMI2W: See Exceptions Type E4NF.nb in Table 2-52, "Type E4NF Class Exception Conditions."

VPERMILPD—Permute In-Lane of Pairs of Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute double precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute double precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
EVEX.128.66.0F38.W1 0D /r VPERMILPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 0D /r VPERMILPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 0D /r VPERMILPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute double precision floating-point values in xmm2/m128 using controls from imm8.
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute double precision floating-point values in ymm2/m256 using controls from imm8.
EVEX.128.66.0F3A.W1 05 /r ib VPERMILPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W1 05 /r ib VPERMILPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	D	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

(Variable control version)

Permute pairs of double precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

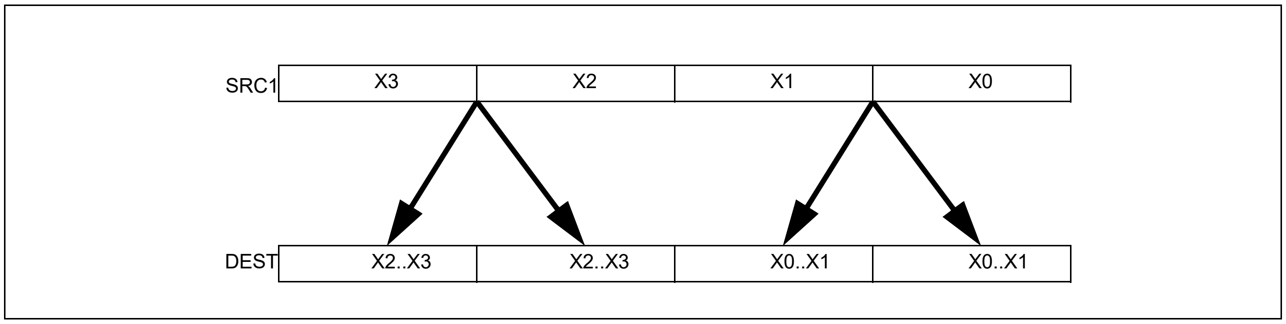


Figure 5-23. VPERMILPD Operation

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

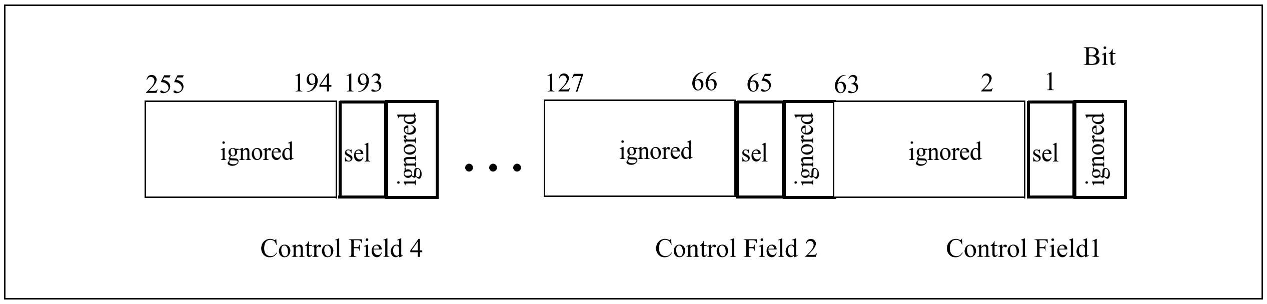


Figure 5-24. VPERMILPD Shuffle Control

Immediate control version: Permute pairs of double precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

Operation

VPERMILPD (EVEX immediate versions)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+63:i] := SRC1[63:0];
        ELSE TMP_SRC1[i+63:i] := SRC1[i+63:i];
    FI;
ENDFOR;
IF (imm8[0] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] := TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] := TMP_SRC1[63:0]; FI;
IF (imm8[1] = 1) THEN TMP_DEST[127:64] := TMP_SRC1[127:64]; FI;
IF VL >= 256
    IF (imm8[2] = 0) THEN TMP_DEST[191:128] := TMP_SRC1[191:128]; FI;
    IF (imm8[2] = 1) THEN TMP_DEST[191:128] := TMP_SRC1[255:192]; FI;
    IF (imm8[3] = 0) THEN TMP_DEST[255:192] := TMP_SRC1[191:128]; FI;
    IF (imm8[3] = 1) THEN TMP_DEST[255:192] := TMP_SRC1[255:192]; FI;
FI;
IF VL >= 512
    IF (imm8[4] = 0) THEN TMP_DEST[319:256] := TMP_SRC1[319:256]; FI;
    IF (imm8[4] = 1) THEN TMP_DEST[319:256] := TMP_SRC1[383:320]; FI;
    IF (imm8[5] = 0) THEN TMP_DEST[383:320] := TMP_SRC1[319:256]; FI;
    IF (imm8[5] = 1) THEN TMP_DEST[383:320] := TMP_SRC1[383:320]; FI;
    IF (imm8[6] = 0) THEN TMP_DEST[447:384] := TMP_SRC1[447:384]; FI;
    IF (imm8[6] = 1) THEN TMP_DEST[447:384] := TMP_SRC1[511:448]; FI;
    IF (imm8[7] = 0) THEN TMP_DEST[511:448] := TMP_SRC1[447:384]; FI;
    IF (imm8[7] = 1) THEN TMP_DEST[511:448] := TMP_SRC1[511:448]; FI;
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPD (256-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] := SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128] := SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128] := SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192] := SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0
```

VPERMILPD (128-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

VPERMILPD (EVEX variable versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;

IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (TMP_SRC2[1] = 1) THEN TMP_DEST[63:0] := SRC1[127:64]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] := SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] := SRC1[127:64]; FI;
IF VL >= 256
    IF (TMP_SRC2[129] = 0) THEN TMP_DEST[191:128] := SRC1[191:128]; FI;
    IF (TMP_SRC2[129] = 1) THEN TMP_DEST[191:128] := SRC1[255:192]; FI;
    IF (TMP_SRC2[193] = 0) THEN TMP_DEST[255:192] := SRC1[191:128]; FI;
    IF (TMP_SRC2[193] = 1) THEN TMP_DEST[255:192] := SRC1[255:192]; FI;
FI;
IF VL >= 512
    IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] := SRC1[319:256]; FI;
    IF (TMP_SRC2[257] = 1) THEN TMP_DEST[319:256] := SRC1[383:320]; FI;
    IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] := SRC1[319:256]; FI;
    IF (TMP_SRC2[321] = 1) THEN TMP_DEST[383:320] := SRC1[383:320]; FI;
    IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] := SRC1[447:384]; FI;
    IF (TMP_SRC2[385] = 1) THEN TMP_DEST[447:384] := SRC1[511:448]; FI;
    IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] := SRC1[447:384]; FI;
    IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] := SRC1[511:448]; FI;
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
```

```
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPD (256-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128] := SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128] := SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192] := SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0
```

VPERMILPD (128-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMILPD __m512d _mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d _mm512_mask_permute_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d _mm512_maskz_permute_pd( __mmask8 k, __m512d a, int imm);
VPERMILPD __m256d _mm256_mask_permute_pd(__m256d  s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d _mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d _mm_mask_permute_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d _mm_maskz_permute_pd( __mmask8 k, __m128d a, int imm);
VPERMILPD __m512d _mm512_permutevar_pd( __m512i i, __m512d a);
VPERMILPD __m512d _mm512_mask_permutevar_pd(__m512d s, __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m512d _mm512_maskz_permutevar_pd( __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m256d _mm256_mask_permutevar_pd(__m256d s, __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m256d _mm256_maskz_permutevar_pd( __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m128d _mm_mask_permutevar_pd(__m128d s, __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d _mm_maskz_permutevar_pd( __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d _mm_permute_pd (__m128d a, int control)
VPERMILPD __m256d _mm256_permute_pd (__m256d a, int control)
VPERMILPD __m128d _mm_permutevar_pd (__m128d a, __m128i control);
VPERMILPD __m256d _mm256_permutevar_pd (__m256d a, __m256i control);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If either (E)VEX.vvvv != 1111B and with imm8.

VPERMILPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute single precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute single precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.
VEX.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute single precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute single precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.
EVEX.128.66.0F38.W0 0C /r VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 0C /r VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 0C /r VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.
EVEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W0 04 /r ibVPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	D	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
D	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Variable control version:

Permute quadruples of single precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-26). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

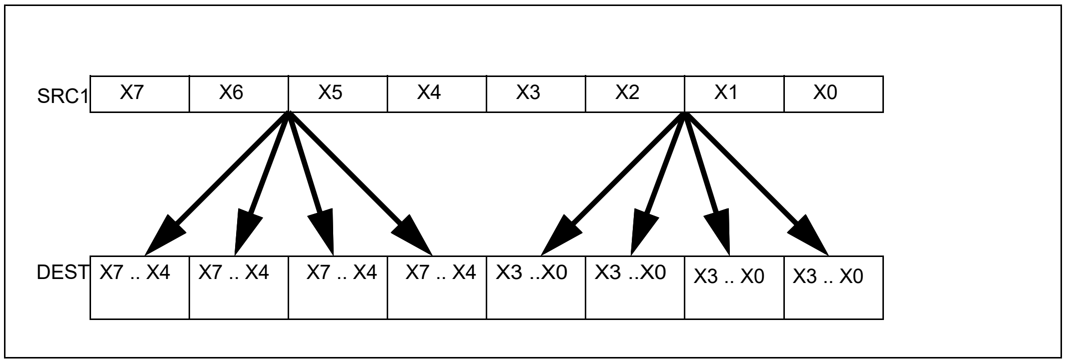


Figure 5-25. VPERMILPS Operation

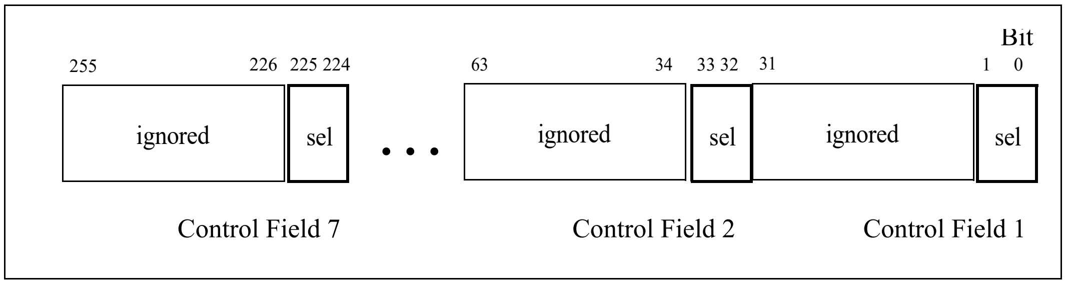


Figure 5-26. VPERMILPS Shuffle Control

(Immediate control version)

Permute quadruples of single precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[31:0];
    1:    TMP := SRC[63:32];
    2:    TMP := SRC[95:64];
    3:    TMP := SRC[127:96];
ESAC;
RETURN TMP
}
```

VPERMILPS (EVEX immediate versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+31:i] := SRC1[31:0];
        ELSE TMP_SRC1[i+31:i] := SRC1[i+31:i];
    FI;
ENDFOR;

TMP_DEST[31:0] := Select4(TMP_SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] := Select4(TMP_SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(TMP_SRC1[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(TMP_SRC1[127:0], imm8[7:6]); FI;
IF VL >= 256
    TMP_DEST[159:128] := Select4(TMP_SRC1[255:128], imm8[1:0]); FI;
    TMP_DEST[191:160] := Select4(TMP_SRC1[255:128], imm8[3:2]); FI;
    TMP_DEST[223:192] := Select4(TMP_SRC1[255:128], imm8[5:4]); FI;
    TMP_DEST[255:224] := Select4(TMP_SRC1[255:128], imm8[7:6]); FI;
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(TMP_SRC1[383:256], imm8[1:0]); FI;
    TMP_DEST[319:288] := Select4(TMP_SRC1[383:256], imm8[3:2]); FI;
    TMP_DEST[351:320] := Select4(TMP_SRC1[383:256], imm8[5:4]); FI;
    TMP_DEST[383:352] := Select4(TMP_SRC1[383:256], imm8[7:6]); FI;
    TMP_DEST[415:384] := Select4(TMP_SRC1[511:384], imm8[1:0]); FI;
    TMP_DEST[447:416] := Select4(TMP_SRC1[511:384], imm8[3:2]); FI;
    TMP_DEST[479:448] := Select4(TMP_SRC1[511:384], imm8[5:4]); FI;
    TMP_DEST[511:480] := Select4(TMP_SRC1[511:384], imm8[7:6]); FI;
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] := 0                ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPS (256-bit immediate version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] := Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] := Select4(SRC1[255:128], imm8[7:6]);
```

VPERMILPS (128-bit immediate version)

```
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0
```

VPERMILPS (EVEX variable versions)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
    FI;
ENDFOR;
TMP_DEST[31:0] := Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] := Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] := Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
    TMP_DEST[159:128] := Select4(SRC1[255:128], TMP_SRC2[129:128]);
    TMP_DEST[191:160] := Select4(SRC1[255:128], TMP_SRC2[161:160]);
    TMP_DEST[223:192] := Select4(SRC1[255:128], TMP_SRC2[193:192]);
    TMP_DEST[255:224] := Select4(SRC1[255:128], TMP_SRC2[225:224]);
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], TMP_SRC2[257:256]);
    TMP_DEST[319:288] := Select4(SRC1[383:256], TMP_SRC2[289:288]);
    TMP_DEST[351:320] := Select4(SRC1[383:256], TMP_SRC2[321:320]);
    TMP_DEST[383:352] := Select4(SRC1[383:256], TMP_SRC2[353:352]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], TMP_SRC2[385:384]);
    TMP_DEST[447:416] := Select4(SRC1[511:384], TMP_SRC2[417:416]);
    TMP_DEST[479:448] := Select4(SRC1[511:384], TMP_SRC2[449:448]);
    TMP_DEST[511:480] := Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] := 0                    ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMILPS (256-bit variable version)

```
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] := Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] := Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] := Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] := Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] := Select4(SRC1[255:128], SRC2[225:224]);
DEST[MAXVL-1:256] := 0
```

VPERMILPS (128-bit variable version)

```
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] :=Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMILPS __m512 _mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 _mm512_mask_permute_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 _mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 _mm256_mask_permute_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 _mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 _mm_mask_permute_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 _mm_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 _mm512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 _mm512_mask_permutevar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 _mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 _mm256_mask_permutevar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 _mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 _mm_mask_permutevar_ps(__m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_maskz_permutevar_ps( __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_permute_ps (__m128 a, int control);
VPERMILPS __m256 _mm256_permute_ps (__m256 a, int control);
VPERMILPS __m128 _mm_permutevar_ps (__m128 a, __m128i control);
VPERMILPS __m256 _mm256_permutevar_ps (__m256 a, __m256i control);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If either (E)VEX.vvvv != 1111B and with imm8.

VPERMPD—Permute Double Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute double precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 01 /r ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F OR AVX10.1	Permute double precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 16 /r VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Permute double precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The imm8 version: Copies quadword elements of double precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPD (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] := SRC[63:0];
        ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
    FI;
ENDFOR;

TMP_DEST[63:0] := (TMP_SRC[256:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] := (TMP_SRC[256:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[191:128] := (TMP_SRC[256:0] >> (IMM8[5:4] * 64))[63:0];
TMP_DEST[255:192] := (TMP_SRC[256:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPD (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
    TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPD (VEX.256 encoded version)

```
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] := (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD __m512d _mm512_permutex_pd(__m512d a, int imm);
VPERMPD __m512d _mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_maskz_permutex_pd(__mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_permutexvar_pd(__m512i i, __m512d a);
VPERMPD __m512d _mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d _mm512_maskz_permutexvar_pd(__mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d _mm256_permutex_epi64(__m256d a, int imm);
VPERMPD __m256d _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_maskz_permutex_epi64(__mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_permutexvar_epi64(__m256i i, __m256d a);
VPERMPD __m256d _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d _mm256_maskz_permutexvar_epi64(__mmask8 k, __m256i i, __m256d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions”; additionally:

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions”; additionally:

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMPS—Permute Single Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 16 /r VPERMPS ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute single precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.256.66.0F38.W0 16 /r VPERMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.
EVEX.512.66.0F38.W0 16 /r VPERMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Copies doubleword elements of single precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPS (EVEX forms)

```
(KL, VL) (8, 256),= (16, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
         THEN TMP_SRC2[i+31:i] := SRC2[31:0];
         ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[31:0] := (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
    TMP_DEST[159:128] := (TMP_SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
FI;
IF VL = 512
    TMP_DEST[31:0] := (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] := (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] := (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] := (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] := (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] := (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] := (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] := (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] :=(TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] := (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMPS (VEX.256 encoded version)

```
DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] := (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] := (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS __m512 _mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 _mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 _mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 _mm256_permutexvar_ps(__m256 i, __m256 a);
VPERMPS __m256 _mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 _mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

VPERMQ—Qwords Element Permutation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.
EVEX.512.66.0F3A.W1 00 /r ib VPERMQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F OR AVX10.1	Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1.
EVEX.256.66.0F38.W1 36 /r VPERMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.
EVEX.512.66.0F38.W1 36 /r VPERMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.

Operation

VPERMQ (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] := SRC[63:0];
        ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
    FI;
ENDFOR;
    TMP_DEST[63:0] := (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0                       ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMQ (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
    TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] := (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] := (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0                   ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMQ (VEX.256 encoded version)

```
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] := (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64(__m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i _mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_maskz_permutex_epi64( __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7D /r VPERMT2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 7D /r VPERMT2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AVX512_VBMI) OR AVX10.1	Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 7D /r VPERMT2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI OR AVX10.1	Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
    id := 3;
ELSE IF VL = 256:
    id := 4;
ELSE IF VL = 512:
    id := 5;
FI;
TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
    off := 8*SRC1[j*8 + id: j*8] ;
    IF k1[j] OR *no writemask*:
        DEST[j*8 + 7: j*8] := SRC1[j*8+id+1]? SRC2[off+7:off] : TMP_DEST[off+7:off];
    ELSE IF *zeroing-masking*
        DEST[j*8 + 7: j*8] := 0;
    *ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMT2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_mask_permutex2var_epi8(__m512i a, __mmask64 k, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMT2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_mask_permutex2var_epi8(__m256i a, __mmask32 k, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMT2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_mask_permutex2var_epi8(__m128i a, __mmask16 k, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 7D /r VPERMT2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 7D /r VPERMT2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 7D /r VPERMT2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 7E /r VPERMT2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 7E /r VPERMT2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 7E /r VPERMT2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 7E /r VPERMT2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 7F /r VPERMT2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Permutes 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id := 2
FI;
IF VL = 256
    id := 3
FI;
IF VL = 512
    id := 4
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 16
    off := 16*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i]=SRC1[i+id+1] ? SRC2[off+15:off]
                    : TMP_DEST[off+15:off]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMT2D/VPERMT2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id := 1
FI;
IF VL = 256
    id := 2
FI;
IF VL = 512
    id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 32
    off := 32*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+id+1] ? SRC2[31:0]
                    : TMP_DEST[off+31:off]
            ELSE
                DEST[i+31:i] := SRC1[i+id+1] ? SRC2[off+31:off]
                    : TMP_DEST[off+31:off]
            FI
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPERMT2Q/VPERMT2PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id := 0
FI;
IF VL = 256
    id := 1
FI;
IF VL = 512
    id := 2
FI;
TMP_DEST:= DEST
FOR j := 0 TO KL-1
    i := j * 64
    off := 64*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := SRC1[i+id+1] ? SRC2[63:0]
                        : TMP_DEST[off+63:off]
            ELSE
                DEST[i+63:i] := SRC1[i+id+1] ? SRC2[off+63:off]
                        : TMP_DEST[off+63:off]
            FI
        ELSE
            IF *merging-masking*                        ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                    ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMT2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMT2D __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
```

VPERMT2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMT2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMT2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMT2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMT2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMT2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMT2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMT2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMT2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMT2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMT2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMT2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMT2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMT2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMT2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);

```
VPERMT2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMT2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMT2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMT2D/Q/PS/PD: See Table 2-52, “Type E4NF Class Exception Conditions.”

VPERMT2W: See Exceptions Type E4NF.nb in Table 2-52, “Type E4NF Class Exception Conditions.”

VPEXPANDB/VPEXPANDW—Expand Byte/Word Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, xmm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, ymm2	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2 OR AVX10.1	Expands up to 512 bits of packed byte integer values from zmm2 to zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expands (loads) up to 64 byte integer values or 32 word integer values from the source operand (memory operand) to the destination operand (register operand), based on the active elements determined by the writemask operand.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k := 0
FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
       DEST.byte[j] := SRC.byte[k];
       k := k + 1
       ELSE:
           IF *merging-masking*:
               *DEST.byte[j] remains unchanged*
               ELSE:                    ; zeroing-masking
                   DEST.byte[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPEXPANDW

```
(KL, VL) = (8,128), (16,256), (32, 512)
k := 0
FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
       DEST.word[j] := SRC.word[k];
       k := k + 1
       ELSE:
           IF *merging-masking*:
               *DEST.word[j] remains unchanged*
               ELSE:                    ; zeroing-masking
                   DEST.word[j] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i _mm_mask_expandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m128i _mm_maskz_expandloadu_epi8(__mmask16, const void*);
VPEXPAND __m256i _mm256_mask_expand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i _mm256_maskz_expand_epi8(__mmask32, __m256i);
VPEXPAND __m256i _mm256_mask_expandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m256i _mm256_maskz_expandloadu_epi8(__mmask32, const void*);
VPEXPAND __m512i _mm512_mask_expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz_expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i _mm512_mask_expandloadu_epi8(__m512i, __mmask64, const void*);
VPEXPAND __m512i _mm512_maskz_expandloadu_epi8(__mmask64, const void*);
VPEXPANDW __m128i _mm_mask_expand_epi16(__m128i, __mmask8, __m128i);
VPEXPANDW __m128i _mm_maskz_expand_epi16(__mmask8, __m128i);
VPEXPANDW __m128i _mm_mask_expandloadu_epi16(__m128i, __mmask8, const void*);
VPEXPANDW __m128i _mm_maskz_expandloadu_epi16(__mmask8, const void *);
VPEXPANDW __m256i _mm256_mask_expand_epi16(__m256i, __mmask16, __m256i);
VPEXPANDW __m256i _mm256_maskz_expand_epi16(__mmask16, __m256i);
VPEXPANDW __m256i _mm256_mask_expandloadu_epi16(__m256i, __mmask16, const void*);
VPEXPANDW __m256i _mm256_maskz_expandloadu_epi16(__mmask16, const void*);
VPEXPANDW __m512i _mm512_mask_expand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i _mm512_maskz_expand_epi16(__mmask32, __m512i);
VPEXPANDW __m512i _mm512_mask_expandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m512i _mm512_maskz_expandloadu_epi16(__mmask32, const void*);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPEXPANDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] := SRC[k+31:k];
            k := k + 32
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz_expand_epi32(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPEXPANDQ—Load Sparse Packed Quadword Integer Values From Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}{z}, xmm2/m128	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}{z}, ymm2/m256	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F OR AVX10.1	Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] := SRC[k+63:k];
            k := k + 64
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDQ __m512i _mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m128i _mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_expand_epi64(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword With Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 90 /vsib VPGATHERDD xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 90 /vsib VPGATHERDD ymm1 {k1}, vm32y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 90 /vsib VPGATHERDD zmm1 {k1}, vm32z	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 90 /vsib VPGATHERDQ xmm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 90 /vsib VPGATHERDQ ymm1 {k1}, vm32x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 90 /vsib VPGATHERDQ zmm1 {k1}, vm32y	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into vector zmm1. The elements are specified via the VSIB (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register (zmm1) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VPGATHERDD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        THEN DEST[i+31:i] := MEM[BASE_ADDR +
                        SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*          ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

VPGATHERDQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        THEN DEST[i+63:i] :=
            MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*          ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD __m512i _mm512_i32gather_epi32(__m512i vdx, void * base, int scale);

```
VPGATHERDD __m512i _mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i _mm256_mmask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i _mm_mmask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_i32logather_epi64( __m256i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDQ __m256i _mm256_mmask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i _mm_mmask_i32gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “Type E12 Class Exception Conditions.”

VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 91 /vsib VPGATHERQQ xmm1 {k1}, vm64x	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 {k1}, vm64y	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 {k1}, vm64z	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A

Description

A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VPGATHERQD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j]
        THEN DEST[i+31:i] := MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+31:i] := remains unchanged*            ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0
```

VPGATHERQQ (EVEX encoded version)

```
(KL, VL) = (2, 64), (4, 128), (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        THEN DEST[i+63:i] :=
            MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] := 0
        ELSE *DEST[i+63:i] := remains unchanged*            ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);

```
VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_i64gather_epi64( __m512i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 44 /r VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 44 /r VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD OR AVX10.1	Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 44 /r VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 44 /r VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512CD) OR AVX10.1	Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD OR AVX10.1	Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPLZCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            temp := 32
            DEST[i+31:i] := 0
            WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
                temp := temp – 1
                DEST[i+31:i] := DEST[i+31:i] + 1
            OD
        ELSE
          IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
            ELSE DEST[i+31:i] := 0
          FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPLZCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            temp := 64
            DEST[i+63:i] := 0
            WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
                temp := temp – 1
                DEST[i+63:i] := DEST[i+63:i] + 1
            OD
        ELSE
          IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
            ELSE DEST[i+63:i] := 0
          FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add High 52-Bit Products to 64-Bit Accumulators

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1, xmm2, xmm3/m128	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1.
VEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1, ymm2, ymm3/m256	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1.
EVEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_IFMA OR AVX10.1	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation

```
VPMADDHUQ srcdest, src1, src2 (VEX version)
VL = (128,256)
KL = VL/64

FOR i in 0 .. KL-1:
    temp128 := zeroextend64(src1.qword[i][51:0]) *zeroextend64(src2.qword[i][51:0])
    srcdest.qword[i] := srcdest.qword[i] +zeroextend64(temp128[103:52])
srcdest[MAXVL:VL] := 0

VPMADD52HUQ (EVEX encoded)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64;
    IF k1[j] OR *no writemask* THEN
        IF src2 is Memory AND EVEX.b=1 THEN
            tsrc2[63:0] := ZeroExtend64(src2[51:0]);
        ELSE
            tsrc2[63:0] := ZeroExtend64(src2[i+51:i];
        FI;
        Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
        Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[103:52]) ;
        DEST[i+63:i] := Temp2[63:0];
    ELSE
        IF *zeroing-masking* THEN
            DEST[i+63:i] := 0;
        ELSE *merge-masking*
            DEST[i+63:i] is unchanged;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMADD52HUQ __m128i _mm_madd52hi_avx_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52HUQ __m128i _mm_maskz_madd52hi_epu64( __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_madd52hi_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52HUQ __m128i _mm_madd52hi_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_mask_madd52hi_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m256i _mm256_madd52hi_avx_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52HUQ __m256i _mm256_mask_madd52hi_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_maskz_madd52hi_epu64( __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m512i _mm512_madd52hi_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_mask_madd52hi_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_maskz_madd52hi_epu64( __mmask8 k, __m512i a, __m512i b, __m512i c);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1, xmm2, xmm3/m128	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1.
VEX.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1, ymm2, ymm3/m256	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1.
EVEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_IFMA AND AVX512VL) OR AVX10.1	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	B	V/V	AVX512_IFMA OR AVX10.1	Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m(r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

Operation

VPMADDLUQ srcdest, src1, src2 (VEX version)

```
VL = (128,256)
KL = VL/64

FOR i in 0 .. KL-1:
    temp128 := zeroextend64(src1.qword[i][51:0]) *zeroextend64(src2.qword[i][51:0])
    srcdest.qword[i] := srcdest.qword[i] +zeroextend64(temp128[51:0])
srcdest[MAXVL:VL] := 0
```

VPMADD52LUQ (EVEX encoded)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64;
    IF k1[j] OR *no writemask* THEN
        IF src2 is Memory AND EVEX.b=1 THEN
            tsrc2[63:0] := ZeroExtend64(src2[51:0]);
        ELSE
            tsrc2[63:0] := ZeroExtend64(src2[i+51:i];
        FI;
        Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
        Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[51:0]) ;
        DEST[i+63:i] := Temp2[63:0];
    ELSE
        IF *zeroing-masking* THEN
            DEST[i+63:i] := 0;
        ELSE *merge-masking*
            DEST[i+63:i] is unchanged;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMADD52LUQ __m128i _mm_madd52lo_avx_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52LUQ __m128i _mm_madd52lo_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm_madd52lo_epu64 (__m128i __X, __m128i __Y, __m128i __Z);
VPMADD52LUQ __m128i _mm_mask_madd52lo_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm_maskz_madd52lo_epu64( __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m256i _mm256_madd52lo_avx_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52LUQ __m256i _mm256_madd52lo_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_madd52lo_epu64 (__m256i __X, __m256i __Y, __m256i __Z);
VPMADD52LUQ __m256i _mm256_mask_madd52lo_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_maskz_madd52lo_epu64( __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m512i _mm512_madd52lo_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_mask_madd52lo_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_maskz_madd52lo_epu64( __mmask8 k, __m512i a, __m512i b, __m512i c);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-51, “Type E4 Class Exception Conditions.”

VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.
EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.
EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1	RM	V/V	AVX512BW OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.
EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.
EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.
EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1	RM	V/V	AVX512BW OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.
EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.
EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.
EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1	RM	V/V	AVX512DQ OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.
EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.
EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.
EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1	RM	V/V	AVX512DQ OR AVX10.1	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVB2M (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF SRC[i+7]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVW2M (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF SRC[i+15]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVD2M (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF SRC[i+31]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPMOVQ2M (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF SRC[i+63]
        THEN    DEST[j] := 1
        ELSE    DEST[j] := 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMPOVB2M __mmask64 _mm512_movepi8_mask( __m512i );
VPMPOVD2M __mmask16 _mm512_movepi32_mask( __m512i );
VPMPOVQ2M __mmask8 _mm512_movepi64_mask( __m512i );
VPMPOVW2M __mmask32 _mm512_movepi16_mask( __m512i );
VPMPOVB2M __mmask32 _mm256_movepi8_mask( __m256i );
VPMPOVD2M __mmask8 _mm256_movepi32_mask( __m256i );
VPMPOVQ2M __mmask8 _mm256_movepi64_mask( __m256i );
VPMPOVW2M __mmask16 _mm256_movepi16_mask( __m256i );
VPMPOVB2M __mmask16 _mm_movepi8_mask( __m128i );
VPMPOVD2M __mmask8 _mm_movepi32_mask( __m128i );
VPMPOVQ2M __mmask8 _mm_movepi64_mask( __m128i );
VPMPOVW2M __mmask8 _mm_movepi16_mask( __m128i );
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-57, “Type E7NM Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 31 /r VPMOVDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed double-word integers from xmm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 21 /r VPMOVSDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed double-word integers from xmm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 31 /r VPMOVDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed double-word integers from ymm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 21 /r VPMOVSDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed signed double-word integers from ymm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 31 /r VPMOVDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 21 /r VPMOVSDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVDB down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB convert unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VPMOVSDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVSDB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateSignedDoubleWordToByte (SRC[m+31:m])
            ELSE *DEST[i+7:i] remains unchanged*            ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSDB instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
            ELSE
                IF *merging-masking*                        ; merging-masking
                    THEN *DEST[i+7:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+7:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/4] := 0;
```

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
            ELSE *DEST[i+7:i] remains unchanged*            ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB void _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_cvtsepi32_epi8(__m512i a);
VPMOVSDB __m128i _mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_maskz_cvtsepi32_epi8(__mmask16 k, __m512i a);
VPMOVSDB void _mm512_mask_cvtsepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_cvtusepi32_epi8(__m512i a);
VPMOVUSDB __m128i _mm512_mask_cvtusepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_maskz_cvtusepi32_epi8(__mmask16 k, __m512i a);
VPMOVUSDB void _mm512_mask_cvtusepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm256_cvtusepi32_epi8(__m256i a);
VPMOVUSDB __m128i _mm256_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVUSDB void _mm256_mask_cvtusepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm_cvtusepi32_epi8(__m128i a);
VPMOVUSDB __m128i _mm_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDB __m128i _mm_maskz_cvtusepi32_epi8(__mmask8 k, __m128i b);
VPMOVUSDB void _mm_mask_cvtusepi32_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm256_cvtsepi32_epi8(__m256i a);
VPMOVSDB __m128i _mm256_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm256_maskz_cvtsepi32_epi8(__mmask8 k, __m256i b);
VPMOVSDB void _mm256_mask_cvtsepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm_cvtsepi32_epi8(__m128i a);
VPMOVSDB __m128i _mm_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm_maskz_cvtsepi32_epi8(__mmask8 k, __m128i b);
VPMOVSDB void _mm_mask_cvtsepi32_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVDB __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVDB __m128i _mm256_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVDB __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVDB void _mm256_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVDB __m128i _mm_cvtepi32_epi8(__m128i a);
VPMOVDB __m128i _mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVDB __m128i _mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i b);
VPMOVDB void _mm_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, "Type E6 Class Exception Conditions."

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 33 /r VPMOVDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed double-word integers from xmm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 23 /r VPMOVSDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed double-word integers from xmm2 into 4 packed signed word integers in ymm1/m64 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 33 /r VPMOVDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed double-word integers from ymm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 23 /r VPMOVSDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed signed double-word integers from ymm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 33 /r VPMOVDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed double-word integers from zmm2 into 16 packed word integers in ymm1/m256 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed signed double-word integers from zmm2 into 16 packed signed word integers in ymm1/m256 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned word integers in ymm1/m256 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVSDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSDW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
            ELSE
                *DEST[i+15:i] remains unchanged*      ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSDW instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+15:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSDW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
            ELSE
                *DEST[i+15:i] remains unchanged*      ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_cvtsepi32_epi16(__m512i a);
VPMOVSDW __m256i _mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSDW void _mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_cvtusepi32_epi16 __m512i a);
VPMOVUSDW __m256i _mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void _mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m128i _mm256_cvtusepi32_epi16(__m256i a);
VPMOVUSDW __m128i _mm256_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i b);
VPMOVUSDW void _mm256_mask_cvtusepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm_cvtusepi32_epi16(__m128i a);
VPMOVUSDW __m128i _mm_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDW __m128i _mm_maskz_cvtusepi32_epi16(__mmask8 k, __m128i b);
VPMOVUSDW void _mm_mask_cvtusepi32_storeu_epi16(void * , __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm256_cvtsepi32_epi16(__m256i a);
VPMOVSDW __m128i _mm256_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i b);
VPMOVSDW void _mm256_mask_cvtsepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm_cvtsepi32_epi16(__m128i a);
VPMOVSDW __m128i _mm_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm_maskz_cvtsepi32_epi16(__mmask8 k, __m128i b);
VPMOVSDW void _mm_mask_cvtsepi32_storeu_epi16(void * , __mmask8 k, __m128i b);
VPMOVDW __m128i _mm256_cvtepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVDW __m128i _mm_cvtepi32_epi16(__m128i a);
VPMOVDW __m128i _mm_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void _mm_mask_cvtepi32_storeu_epi16(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each byte in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each byte in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1	RM	V/V	AVX512BW OR AVX10.1	Sets each byte in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each word in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1	RM	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Sets each word in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1	RM	V/V	AVX512BW OR AVX10.1	Sets each word in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each doubleword in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each doubleword in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1	RM	V/V	AVX512DQ OR AVX10.1	Sets each doubleword in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each quadword in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1	RM	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Sets each quadword in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1	RM	V/V	AVX512DQ OR AVX10.1	Sets each quadword in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1’s or all 0’s depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVM2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF SRC[j]
        THEN    DEST[i+7:i] := -1
        ELSE    DEST[i+7:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF SRC[j]
        THEN    DEST[i+15:i] := -1
        ELSE    DEST[i+15:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2D (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF SRC[j]
        THEN    DEST[i+31:i] := -1
        ELSE    DEST[i+31:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPMOVM2Q (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF SRC[j]
        THEN    DEST[i+63:i] := -1
        ELSE    DEST[i+63:i] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i _mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i _mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i _mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i _mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i _mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i _mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i _mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i _mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i _mm_movm_epi8(__mmask16);
VPMOVM2D __m128i _mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i _mm_movm_epi64(__mmask8);
VPMOVM2W __m128i _mm_movm_epi16(__mmask8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-57, “Type E7NM Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 32 /r VPMOVQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed quad-word integers from xmm2 into 2 packed byte integers in xmm1/m16 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 22 /r VPMOVSQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed byte integers in xmm1/m16 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m16 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned byte integers in xmm1/m16 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 32 /r VPMOVQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed quad-word integers from ymm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 22 /r VPMOVSQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m32 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 32 /r VPMOVQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed quad-word integers from zmm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m64 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Eighth Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;
```

VPMOVQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            *DEST[i+7:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVSQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 8
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;
```

VPMOVSQB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
            ELSE
                *DEST[i+7:i] remains unchanged*             ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
            ELSE
                IF *merging-masking*                        ; merging-masking
                    THEN *DEST[i+7:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+7:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/8] := 0;
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 8
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
            ELSE
                *DEST[i+7:i] remains unchanged*             ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_cvtsepi64_epi8(__m512i a);
VPMOVSQB __m128i _mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtsepi64_epi8(__mmask8 k, __m512i a);
VPMOVSQB void _mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_cvtusepi64_epi8(__m512i a);
VPMOVUSQB __m128i _mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_maskz_cvtusepi64_epi8(__mmask8 k, __m512i a);
VPMOVUSQB void _mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm256_cvtusepi64_epi8(__m256i a);
VPMOVUSQB __m128i _mm256_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm256_maskz_cvtusepi64_epi8(__mmask8 k, __m256i b);
VPMOVUSQB void _mm256_mask_cvtusepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm_cvtusepi64_epi8(__m128i a);
VPMOVUSQB __m128i _mm_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQB __m128i _mm_maskz_cvtusepi64_epi8(__mmask8 k, __m128i b);
VPMOVUSQB void _mm_mask_cvtusepi64_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm256_cvtsepi64_epi8(__m256i a);
VPMOVSQB __m128i _mm256_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm256_maskz_cvtsepi64_epi8(__mmask8 k, __m256i b);
VPMOVSQB void _mm256_mask_cvtsepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm_cvtsepi64_epi8(__m128i a);
VPMOVSQB __m128i _mm_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm_maskz_cvtsepi64_epi8(__mmask8 k, __m128i b);
VPMOVSQB void _mm_mask_cvtsepi64_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVQB __m128i _mm256_cvtepi64_epi8(__m256i a);
VPMOVQB __m128i _mm256_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256_maskz_cvtepi64_epi8(__mmask8 k, __m256i b);
VPMOVQB void _mm256_mask_cvtepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVQB __m128i _mm_cvtepi64_epi8(__m128i a);
VPMOVQB __m128i _mm_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVQB __m128i _mm_maskz_cvtepi64_epi8(__mmask8 k, __m128i b);
VPMOVQB void _mm_mask_cvtepi64_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed quad-word integers from xmm2 into 2 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.
EVEX.128.F3.0F38.W0 25 /r VPMOVSQD xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed double-word integers in xmm1/m64 using signed saturation subject to writemask k1.
EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned double-word integers in xmm1/m64 using unsigned saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed quad-word integers from ymm2 into 4 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.
EVEX.256.F3.0F38.W0 25 /r VPMOVSQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed double-word integers in xmm1/m128 using signed saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned double-word integers in xmm1/m128 using unsigned saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 35 /r VPMOVQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed quad-word integers from zmm2 into 8 packed double-word integers in ymm1/m256 with truncation subject to writemask k1.
EVEX.512.F3.0F38.W0 25 /r VPMOVSQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed double-word integers in ymm1/m256 using signed saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned double-word integers in ymm1/m256 using unsigned saturation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *zeroing-masking*                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*         ; merging-masking
    FI;
ENDFOR
```

VPMOVSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                  ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
```

```
    DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSQD instruction (EVEX encoded version) memory form

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 32
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
            ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSQD instruction (EVEX encoded version) reg-reg form

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 32
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
            ELSE
                IF *merging-masking*                       ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*                 ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSQD instruction (EVEX encoded version) memory form

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 32
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
            ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMOVQD __m256i _mm512_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD void _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_cvtsepi64_epi32( __m512i a);
VPMOVSQD __m256i _mm512_mask_cvtsepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_maskz_cvtsepi64_epi32( __mmask8 k, __m512i a);
VPMOVSQD void _mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_cvtusepi64_epi32( __m512i a);
VPMOVUSQD __m256i _mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_maskz_cvtusepi64_epi32( __mmask8 k, __m512i a);
VPMOVUSQD void _mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32( __mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32( __mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32( __mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32( __mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 34 /r VPMOVQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed quad-word integers from xmm2 into 2 packed word integers in xmm1/m32 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 24 /r VPMOVSQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m32 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m32 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned word integers in xmm1/m32 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 34 /r VPMOVQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed quad-word integers from ymm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 24 /r VPMOVSQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m64 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 34 /r VPMOVQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed quad-word integers from zmm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 24 /r VPMOVSQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m128 {k1}{z}, zmm2	A	V/V	AVX512F OR AVX10.1	Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVSQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 16
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

VPMOVSQW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
            ELSE
                *DEST[i+15:i] remains unchanged*      ; merging-masking
        FI;
    ENDFOR
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is a register

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
            ELSE
                IF *merging-masking*                  ; merging-masking
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*            ; zeroing-masking
                        DEST[i+15:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL/4] := 0;
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is memory

```
    (KL, VL) = (2, 128), (4, 256), (8, 512)
    FOR j := 0 TO KL-1
        i := j * 16
        m := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
            ELSE
                *DEST[i+15:i] remains unchanged*      ; merging-masking
        FI;
    ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW __m128i _mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtusepi64_epi16(__m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtusepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtusepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 30 /r VPMOVWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 8 packed word integers from xmm2 into 8 packed bytes in xmm1/m64 with truncation under writemask k1.
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 8 packed signed word integers from xmm2 into 8 packed signed bytes in xmm1/m64 using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m64 {k1}{z}, xmm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in 8mm1/m64 using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 30 /r VPMOVWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 16 packed word integers from ymm2 into 16 packed bytes in xmm1/m128 with truncation under writemask k1.
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 16 packed signed word integers from ymm2 into 16 packed signed bytes in xmm1/m128 using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m128 {k1}{z}, ymm2	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Converts 16 packed unsigned word integers from ymm2 into 16 packed unsigned bytes in xmm1/m128 using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 30 /r VPMOVWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1	Converts 32 packed word integers from zmm2 into 32 packed bytes in ymm1/m256 with truncation under writemask k1.
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1	Converts 32 packed signed word integers from zmm2 into 32 packed signed bytes in ymm1/m256 using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB ymm1/m256 {k1}{z}, zmm2	A	V/V	AVX512BW OR AVX10.1	Converts 32 packed unsigned word integers from zmm2 into 32 packed unsigned bytes in ymm1/m256 using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*         ; merging-masking
    FI;
ENDFOR
```

VPMOVSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                   ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*             ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Kl-1
    i := j * 8
    m := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMOVUSWB __m256i _mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i _mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i _mm512_maskz_cvtusepi16_epi8( __mmask32 k, __m512i b);
VPMOVUSWB void _mm512_mask_cvtusepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_cvtsepi16_epi8(__m512i a);
VPMOVSWB __m256i _mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_maskz_cvtsepi16_epi8( __mmask32 k, __m512i b);
VPMOVSWB void _mm512_mask_cvtsepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_cvtepi16_epi8(__m512i a);
VPMOVWB __m256i _mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_maskz_cvtepi16_epi8( __mmask32 k, __m512i b);
VPMOVWB void _mm512_mask_cvtepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVUSWB __m128i _mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSWB __m128i _mm256_mask_cvtusepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm256_maskz_cvtusepi16_epi8( __mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtusepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtusepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtusepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtusepi16_epi8( __mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtusepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm256_cvtsepi16_epi8(__m256i a);
VPMOVSWB __m128i _mm256_mask_cvtsepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm256_maskz_cvtsepi16_epi8( __mmask16 k, __m256i b);
VPMOVSWB void _mm256_mask_cvtsepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm_cvtsepi16_epi8(__m128i a);
VPMOVSWB __m128i _mm_mask_cvtsepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm_maskz_cvtsepi16_epi8( __mmask8 k, __m128i b);
VPMOVSWB void _mm_mask_cvtsepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVWB __m128i _mm256_cvtepi16_epi8(__m256i a);
VPMOVWB __m128i _mm256_mask_cvtepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVWB __m128i _mm256_maskz_cvtepi16_epi8( __mmask16 k, __m256i b);
VPMOVWB void _mm256_mask_cvtepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVWB __m128i _mm_cvtepi16_epi8(__m128i a);
VPMOVWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVWB __m128i _mm_maskz_cvtepi16_epi8( __mmask8 k, __m128i b);
VPMOVWB void _mm_mask_cvtepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E6 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VPMULTISHIFTQB—Select Packed Unaligned Bytes From Quadword Sources

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512_VBMI) OR AVX10.1	Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1.
EVEX.256.66.0F38.W1 83 /r VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst	A	V/V	(AVX512VL AVX512_VBMI) OR AVX10.1	Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1.
EVEX.512.66.0F38.W1 83 /r VPMULTISHIFTQB zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_VBMI OR AVX10.1	Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is extracted from the input qword source at the location specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.

Operation

```
VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)
(KL, VL) = (2, 128),(4, 256), (8, 512)
FOR i := 0 TO KL-1
    IF EVEX.b=1 AND src2 is memory THEN
            tcur := src2.qword[0]; //broadcasting
    ELSE
            tcur := src2.qword[i];
    FI;
    FOR j := 0 to 7
        ctrl := src1.qword[i].byte[j] & 63;
        FOR k := 0 to 7
            res.bit[k] := tcur.bit[ (ctrl+k) mod 64 ];
        ENDFOR
        IF k1[i*8+j] or no writemask THEN
            DEST.qword[i].byte[j] := res;
        ELSE IF zeroing-masking THEN
            DEST.qword[i].byte[j] := 0;
    ENDFOR
ENDFOR
DEST.qword[MAX_VL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULTISHIFTQB __m512i _mm512_multishift_epi64_epi8( __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_mask_multishift_epi64_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_maskz_multishift_epi64_epi8( __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m256i _mm256_multishift_epi64_epi8( __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_mask_multishift_epi64_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_maskz_multishift_epi64_epi8( __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m128i _mm_multishift_epi64_epi8( __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_mask_multishift_epi64_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_maskz_multishift_epi64_epi8( __mmask8 k, __m128i a, __m128i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-52, "Type E4NF Class Exception Conditions."

VPOPCNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 54 /r VPOPCNTB xmm1{k1}{z}, xmm2/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 54 /r VPOPCNTB ymm1{k1}{z}, ymm2/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 54 /r VPOPCNTB zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG OR AVX10.1	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 54 /r VPOPCNTW xmm1{k1}{z}, xmm2/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 54 /r VPOPCNTW ymm1{k1}{z}, ymm2/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 54 /r VPOPCNTW zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG OR AVX10.1	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W0 55 /r VPOPCNTD xmm1{k1}{z}, xmm2/m128/m32bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 55 /r VPOPCNTD ymm1{k1}{z}, ymm2/m256/m32bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 55 /r VPOPCNTD zmm1{k1}{z}, zmm2/m512/m32bcst	B	V/V	AVX512_VPOPCNTDQ OR AVX10.1	Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 55 /r VPOPCNTQ xmm1{k1}{z}, xmm2/m128/m64bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 55 /r VPOPCNTQ ymm1{k1}{z}, ymm2/m256/m64bcst	B	V/V	(AVX512_VPOPCNTDQ AND AVX512VL) OR AVX10.1	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 55 /r VPOPCNTQ zmm1{k1}{z}, zmm2/m512/m64bcst	B	V/V	AVX512_VPOPCNTDQ OR AVX10.1	Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction counts the number of bits set to one in each byte, word, dword or qword element of its source (e.g., zmm2 or memory) and places the results in the destination register (zmm1). This instruction supports memory fault suppression.

Operation

VPOPCNTB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.byte[j] := POPCNT(SRC.byte[j])
    ELSE IF *merging-masking*:
        *DEST.byte[j] remains unchanged*
    ELSE:
        DEST.byte[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := POPCNT(SRC.word[j])
    ELSE IF *merging-masking*:
        *DEST.word[j] remains unchanged*
    ELSE:
        DEST.word[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t := SRC.dword[0]
        ELSE:
            t := SRC.dword[j]
        DEST.dword[j] := POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..dword[j] remains unchanged*
    ELSE:
        DEST..dword[j] := 0
DEST[MAX_VL-1:VL] := 0
```

VPOPCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t := SRC.qword[0]
        ELSE:
            t := SRC.qword[j]
        DEST.qword[j] := POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..qword[j] remains unchanged*
    ELSE:
        DEST..qword[j] := 0
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i  _mm512_maskz_popcnt_epi16(__mmask32, __m512i);
VPOPCNTQ __m128i  _mm_popcnt_epi64(__m128i);
VPOPCNTQ __m128i  _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTQ __m128i  _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i  _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i  _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i  _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i  _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i  _mm512_mask_popcnt_epi64(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i  _mm512_maskz_popcnt_epi64(__mmask8, __m512i);
VPOPCNTD __m128i  _mm_popcnt_epi32(__m128i);
VPOPCNTD __m128i  _mm_mask_popcnt_epi32(__m128i, __mmask8, __m128i);
VPOPCNTD __m128i  _mm_maskz_popcnt_epi32(__mmask8, __m128i);
VPOPCNTD __m256i  _mm256_popcnt_epi32(__m256i);
VPOPCNTD __m256i  _mm256_mask_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTD __m256i  _mm256_maskz_popcnt_epi32(__mmask8, __m256i);
VPOPCNTD __m512i  _mm512_popcnt_epi32(__m512i);
VPOPCNTD __m512i  _mm512_mask_popcnt_epi32(__m512i, __mmask16, __m512i);
VPOPCNTD __m512i  _mm512_maskz_popcnt_epi32(__mmask16, __m512i);
VPOPCNTB __m128i  _mm_popcnt_epi8(__m128i);
VPOPCNTB __m128i  _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTB __m128i  _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTB __m256i  _mm256_popcnt_epi8(__m256i);
VPOPCNTB __m256i  _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i  _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i  _mm512_popcnt_epi8(__m512i);
VPOPCNTB __m512i  _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i  _mm512_maskz_popcnt_epi8(__mmask64, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 15 /r VPROLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.
EVEX.128.66.0F.W0 72 /1 ib VPROLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.128.66.0F38.W1 15 /r VPROLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.
EVEX.128.66.0F.W1 72 /1 ib VPROLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 15 /r VPROLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.
EVEX.256.66.0F.W0 72 /1 ib VPROLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.256.66.0F38.W1 15 /r VPROLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.
EVEX.256.66.0F.W1 72 /1 ib VPROLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 15 /r VPROLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1[1]	Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.
EVEX.512.66.0F.W0 72 /1 ib VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate left of doublewords in zmm3/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.
EVEX.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1under writemask k1.
EVEX.512.66.0F.W1 72 /1 ib VPROLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	imm8	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC << COUNT) | (SRC >> (64 - COUNT));
```

VPROLD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
          IF (EVEX.b = 1) AND (SRC1 *is memory*)
             THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[31:0], imm8)
             ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
          FI;
      ELSE
          IF *merging-masking*                  ; merging-masking
             THEN *DEST[i+31:i] remains unchanged*
             ELSE *zeroing-masking*              ; zeroing-masking
                DEST[i+31:i] := 0
          FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPROLVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
                ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPROLQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPROLVQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
                ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPROLD __m512i _mm512_rol_epi32(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi32( __mmask16 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi32(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi32( __mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rol_epi32(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi32( __mmask8 k, __m128i a, int imm);
VPROLQ __m512i _mm512_rol_epi64(__m512i a, int imm);
VPROLQ __m512i _mm512_mask_rol_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPROLQ __m512i _mm512_maskz_rol_epi64(__mmask8 k, __m512i a, int imm);
VPROLQ __m256i _mm256_rol_epi64(__m256i a, int imm);
VPROLQ __m256i _mm256_mask_rol_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLQ __m256i _mm256_maskz_rol_epi64( __mmask8 k, __m256i a, int imm);
VPROLQ __m128i _mm_rol_epi64(__m128i a, int imm);
VPROLQ __m128i _mm_mask_rol_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLQ __m128i _mm_maskz_rol_epi64( __mmask8 k, __m128i a, int imm);
VPROLVD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLVD __m512i _mm512_mask_rolv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPROLVD __m512i _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPROLVD __m256i _mm256_rolv_epi32(__m256i a, __m256i cnt);
VPROLVD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLVD __m128i _mm_rolv_epi32(__m128i a, __m128i cnt);
VPROLVD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPROLVQ __m512i _mm512_rolv_epi64(__m512i a, __m512i cnt);
VPROLVQ __m512i _mm512_mask_rolv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPROLVQ __m512i _mm512_maskz_rolv_epi64( __mmask8 k, __m512i a, __m512i cnt);
VPROLVQ __m256i _mm256_rolv_epi64(__m256i a, __m256i cnt);
VPROLVQ __m256i _mm256_mask_rolv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVQ __m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPROLVQ __m128i _mm_rolv_epi64(__m128i a, __m128i cnt);
VPROLVQ __m128i _mm_mask_rolv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVQ __m128i _mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i cnt);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 14 /r VPRORVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.
EVEX.128.66.0F.W0 72 /0 ib VPRORD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.
EVEX.128.66.0F38.W1 14 /r VPRORVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.
EVEX.128.66.0F.W1 72 /0 ib VPRORQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.
EVEX.256.66.0F38.W0 14 /r VPRORVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1.
EVEX.256.66.0F.W0 72 /0 ib VPRORD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.
EVEX.256.66.0F38.W1 14 /r VPRORVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.
EVEX.256.66.0F.W1 72 /0 ib VPRORQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.
EVEX.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1[1]	Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.
EVEX.512.66.0F.W0 72 /0 ib VPRORD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.
EVEX.512.66.0F38.W1 14 /r VPRORVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.
EVEX.512.66.0F.W1 72 /0 ib VPRORQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	imm8	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC >> COUNT) | (SRC << (64 - COUNT));
```

VPRORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
           IF (EVEX.b = 1) AND (SRC1 *is memory*)
               THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
               ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
           FI;
       ELSE
           IF *merging-masking*                ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
               ELSE *zeroing-masking*           ; zeroing-masking
                   DEST[i+31:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPRORVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
           IF (EVEX.b = 1) AND (SRC2 *is memory*)
               THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
               ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
           FI;
```

```
            ELSE
                IF *merging-masking*            ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*            ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPRORQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC1 *is memory*)
                    THEN DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[63:0], imm8)
                    ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], imm8])
                FI;
            ELSE
                IF *merging-masking*            ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*            ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VPRORVQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
                    ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
                FI;
            ELSE
                IF *merging-masking*            ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*            ; zeroing-masking
                        DEST[i+63:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32( __mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32( __mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32( __mmask8 k, __m128i a, int imm);
VPRORQ __m512i _mm512_ror_epi64(__m512i a, int imm);
VPRORQ __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPRORQ __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
VPRORQ __m256i _mm256_ror_epi64(__m256i a, int imm);
VPRORQ __m256i _mm256_mask_ror_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORQ __m256i _mm256_maskz_ror_epi64( __mmask8 k, __m256i a, int imm);
VPRORQ __m128i _mm_ror_epi64(__m128i a, int imm);
VPRORQ __m128i _mm_mask_ror_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORQ __m128i _mm_maskz_ror_epi64( __mmask8 k, __m128i a, int imm);
VPRORVD __m512i _mm512_rorv_epi32(__m512i a, __m512i cnt);
VPRORVD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORVD __m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORVD __m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORVD __m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVD __m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORVD __m128i _mm_rorv_epi32(__m128i a, __m128i cnt);
VPRORVD __m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVD __m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPRORVQ __m512i _mm512_rorv_epi64(__m512i a, __m512i cnt);
VPRORVQ __m512i _mm512_mask_rorv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORVQ __m512i _mm512_maskz_rorv_epi64( __mmask8 k, __m512i a, __m512i cnt);
VPRORVQ __m256i _mm256_rorv_epi64(__m256i a, __m256i cnt);
VPRORVQ __m256i _mm256_mask_rorv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVQ __m256i _mm256_maskz_rorv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPRORVQ __m128i _mm_rorv_epi64(__m128i a, __m128i cnt);
VPRORVQ __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVQ __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.128.66.0F38.W0 A1 /vsib VPSCATTERQD vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A1 /vsib VPSCATTERQD vm64y {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A1 /vsib VPSCATTERQD vm64z {k1}, ymm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter qword values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	N/A	N/A

Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VPSCATTERDD (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] := SRC[i+31:i]
            k1[j] := 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERDQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
```

```
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] := SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERQD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] := SRC[i+31:i]
            k1[j] := 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VPSCATTERQQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] := SRC[i+63:i]
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_i64scatter_epi32(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_i64scatter_epi64(void * base, __m256i vdx, __m256i a, int scale);
```

VPSCATTERQQ void _mm_i64scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm_mask_i64scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VPSHLD—Concatenate and Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 70 /r /ib VPSHLDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 70 /r /ib VPSHLDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 70 /r /ib VPSHLDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W0 71 /r /ib VPSHLDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W0 71 /r /ib VPSHLDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W0 71 /r /ib VPSHLDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W1 71 /r /ib VPSHLDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 71 /r /ib VPSHLDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 71 /r /ib VPSHLDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the left by constant value.

This instruction supports memory fault suppression.

Operation

VPSHLDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.word[j], SRC3.word[j]) << (imm8 & 15)
        DEST.word[j] := tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.dword[j], tsrc3) << (imm8 & 31)
        DEST.dword[j] := tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(SRC2.qword[j], tsrc3) << (imm8 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDD __m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLDD __m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDD __m128i _mm_maskz_shldi_epi32(__mmask8, __m128i, __m128i, int);
VPSHLDD __m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLDD __m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDD __m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLDD __m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLDD __m512i _mm512_mask_shldi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHLDD __m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLDQ __m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDQ __m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDQ __m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDQ __m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDQ __m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDQ __m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDQ __m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDQ __m512i _mm512_mask_shldi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHLDQ __m512i _mm512_maskz_shldi_epi64(__mmask8, __m512i, __m512i, int);
VPSHLDW __m128i _mm_shldi_epi16(__m128i, __m128i, int);
VPSHLDW __m128i _mm_mask_shldi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDW __m128i _mm_maskz_shldi_epi16(__mmask8, __m128i, __m128i, int);
VPSHLDW __m256i _mm256_shldi_epi16(__m256i, __m256i, int);
VPSHLDW __m256i _mm256_mask_shldi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHLDW __m256i _mm256_maskz_shldi_epi16(__mmask16, __m256i, __m256i, int);
VPSHLDW __m512i _mm512_shldi_epi16(__m512i, __m512i, int);
VPSHLDW __m512i _mm512_mask_shldi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHLDW __m512i _mm512_maskz_shldi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 70 /r VPSHLDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 70 /r VPSHLDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 70 /r VPSHLDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W0 71 /r VPSHLDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W0 71 /r VPSHLDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W0 71 /r VPSHLDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W1 71 /r VPSHLDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 71 /r VPSHLDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 71 /r VPSHLDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenate packed data, extract result shifted to the left by variable value.

This instruction supports memory fault suppression.

Operation

```
FUNCTION concat(a,b):
    IF words:
        d.word[1] := a
        d.word[0] := b
        return d
    ELSE IF dwords:
        q.dword[1] := a
        q.dword[0] := b
        return q
    ELSE IF qwords:
        o.qword[1] := a
        o.qword[0] := b
        return o
```

VPSHLDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.word[j], SRC2.word[j]) << (SRC3.word[j] & 15)
        DEST.word[j] := tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHLDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.dword[j], SRC2.dword[j]) << (tsrc3 & 31)
        DEST.dword[j] := tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

```
VPSHLDVQ DEST, SRC2, SRC3
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.qword[j], SRC2.qword[j]) << (tsrc3 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVQ __m512i _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_mask_shldv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_maskz_shldv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVW __m512i _mm512_shldv_epi16(__m512i, __m512i, __m512i);
VPSHLDVW __m512i _mm512_mask_shldv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHLDVW __m512i _mm512_maskz_shldv_epi16(__mmask32, __m512i, __m512i, __m512i);
VPSHLDVD __m128i _mm_shldv_epi32(__m128i, __m128i, __m128i);
VPSHLDVD __m128i _mm_mask_shldv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVD __m128i _mm_maskz_shldv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m256i _mm256_shldv_epi32(__m256i, __m256i, __m256i);
VPSHLDVD __m256i _mm256_mask_shldv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHLDVD __m256i _mm256_maskz_shldv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHLDVD __m512i _mm512_shldv_epi32(__m512i, __m512i, __m512i);
VPSHLDVD __m512i _mm512_mask_shldv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHLDVD __m512i _mm512_maskz_shldv_epi32(__mmask16, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPSHRD—Concatenate and Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 72 /r /ib VPSHRDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 72 /r /ib VPSHRDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 72 /r /ib VPSHRDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W0 73 /r /ib VPSHRDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W0 73 /r /ib VPSHRDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W0 73 /r /ib VPSHRDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.128.66.0F3A.W1 73 /r /ib VPSHRDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.256.66.0F3A.W1 73 /r /ib VPSHRDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.512.66.0F3A.W1 73 /r /ib VPSHRDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the right by constant value.

This instruction supports memory fault suppression.

Operation

VPSHRDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := concat(SRC3.word[j], SRC2.word[j]) >> (imm8 & 15)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] := concat(tsrc3, SRC2.dword[j]) >> (imm8 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] := concat(tsrc3, SRC2.qword[j]) >> (imm8 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHRDQ __m128i  _mm_shrdi_epi64(__m128i, __m128i, int);
VPSHRDQ __m128i  _mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i  _mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i  _mm256_shrdi_epi64(__m256i, __m256i, int);
VPSHRDQ __m256i  _mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDQ __m256i  _mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPSHRDQ __m512i  _mm512_shrdi_epi64(__m512i, __m512i, int);
VPSHRDQ __m512i  _mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHRDQ __m512i  _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);
VPSHRDD __m128i _mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDD __m128i _mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDD __m128i _mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDD __m256i _mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDD __m256i _mm256_mask_shrdi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDD __m256i _mm256_maskz_shrdi_epi32(__mmask8, __m256i, __m256i, int);
VPSHRDD __m512i _mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDD __m512i _mm512_mask_shrdi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHRDD __m512i _mm512_maskz_shrdi_epi32(__mmask16, __m512i, __m512i, int);
VPSHRDW __m128i _mm_shrdi_epi16(__m128i, __m128i, int);
VPSHRDW __m128i _mm_mask_shrdi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDW __m128i _mm_maskz_shrdi_epi16(__mmask8, __m128i, __m128i, int);
VPSHRDW __m256i _mm256_shrdi_epi16(__m256i, __m256i, int);
VPSHRDW __m256i _mm256_mask_shrdi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDW __m256i _mm256_maskz_shrdi_epi16(__mmask16, __m256i, __m256i, int);
VPSHRDW __m512i _mm512_shrdi_epi16(__m512i, __m512i, int);
VPSHRDW __m512i _mm512_mask_shrdi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDW __m512i _mm512_maskz_shrdi_epi16(__mmask32, __m512i, __m512i, int);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPSHRDV—Concatenate and Variable Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 72 /r VPSHRDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 72 /r VPSHRDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 72 /r VPSHRDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W0 73 /r VPSHRDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W0 73 /r VPSHRDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W0 73 /r VPSHRDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.128.66.0F38.W1 73 /r VPSHRDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.256.66.0F38.W1 73 /r VPSHRDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512_VBMI2 AND AVX512VL) OR AVX10.1	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.512.66.0F38.W1 73 /r VPSHRDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2 OR AVX10.1	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Concatenate packed data, extract result shifted to the right by variable value.

This instruction supports memory fault suppression.

Operation

VPSHRDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] := concat(SRC2.word[j], DEST.word[j]) >> (SRC3.word[j] & 15)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.dword[0]
    ELSE:
        tsrc3 := SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] := concat(SRC2.dword[j], DEST.dword[j]) >> (tsrc3 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] := 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

VPSHRDVQ DEST, SRC2, SRC3

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] := concat(SRC2.qword[j], DEST.qword[j]) >> (tsrc3 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHRDVQ __m128i _mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVD __m128i _mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVD __m128i _mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVD __m128i _mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVD __m256i _mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVD __m256i _mm256_mask_shrdv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVD __m256i _mm256_maskz_shrdv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVD __m512i _mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVD __m512i _mm512_mask_shrdv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHRDVD __m512i _mm512_maskz_shrdv_epi32(__mmask16, __m512i, __m512i, __m512i);
VPSHRDVW __m128i _mm_shrdv_epi16(__m128i, __m128i, __m128i);
VPSHRDVW __m128i _mm_mask_shrdv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVW __m128i _mm_maskz_shrdv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVW __m256i _mm256_shrdv_epi16(__m256i, __m256i, __m256i);
VPSHRDVW __m256i _mm256_mask_shrdv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVW __m256i _mm256_maskz_shrdv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVW __m512i _mm512_shrdv_epi16(__m512i, __m512i, __m512i);
VPSHRDVW __m512i _mm512_mask_shrdv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVW __m512i _mm512_maskz_shrdv_epi16(__mmask32, __m512i, __m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPSHUFBITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, xmm2, xmm3/m128	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Extract values in xmm2 using control bits of xmm3/m128 with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, ymm2, ymm3/m256	A	V/V	(AVX512_BITALG AND AVX512VL) OR AVX10.1	Extract values in ymm2 using control bits of ymm3/m256 with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, zmm2, zmm3/m512	A	V/V	AVX512_BITALG OR AVX10.1	Extract values in zmm2 using control bits of zmm3/m512 with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The VPSHUFBITQMB instruction performs a bit gather select using second source as control and first source as data. Each bit uses 6 control bits (2nd source operand) to select which data bit is going to be gathered (first source operand). A given bit can only access 64 different bits of data (first 64 destination bits can access first 64 data bits, second 64 destination bits can access second 64 data bits, etc.).

Control data for each output bit is stored in 8 bit elements of SRC2, but only the 6 least significant bits of each element are used.

This instruction uses write masking (zeroing only). This instruction supports memory fault suppression.

The first source operand is a ZMM register. The second source operand is a ZMM register or a memory location. The destination operand is a mask register.

Operation

VPSHUFBITQMB DEST, SRC1, SRC2

```
(KL, VL) = (16,128), (32,256), (64, 512)
FOR i := 0 TO KL/8-1:              //Qword
    FOR j := 0 to 7:               // Byte
        IF k2[i*8+j] or *no writemask*:
            m := SRC2.qword[i].byte[j] & 0x3F
            k1[i*8+j] := SRC1.qword[i].bit[m]
        ELSE:
            k1[i*8+j] := 0
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 _mm_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 _mm_mask_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 _mm256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 _mm256_mask_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 _mm512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 _mm512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);

VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.128.66.0F38.W1 12 /r VPSLLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 12 /r VPSLLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 12 /r VPSLLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W0 47 /r VPSLLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W0 47 /r VPSLLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W0 47 /r VPSLLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W1 47 /r VPSLLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 47 /r VPSLLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 47 /r VPSLLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSLLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
      ELSE
         IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE                                 ; zeroing-masking
               DEST[i+15:i] := 0
         FI
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSLLVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] := ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSLLVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] := ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSLLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
                ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSLLVQ (VEX.128 version)

```
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
DEST[127:64] := ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSLLVQ (VEX.256 version)

```
COUNT_0 := SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] := ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSLLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
                ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m256i _mm256_sllv_epi32 (__m256i m, __m256i count)
VPSLLVQ __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPSLLVD/VPSLLVQ, see Table 2-51, "Type E4 Class Exception Conditions."
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb in Table 2-51, "Type E4 Class Exception Conditions."

VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 46 /r VPSRAVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.
VEX.256.66.0F38.W0 46 /r VPSRAVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.
EVEX.128.66.0F38.W1 11 /r VPSRAVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F38.W1 11 /r VPSRAVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.
EVEX.512.66.0F38.W1 11 /r VPSRAVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.
EVEX.128.66.0F38.W0 46 /r VPSRAVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.
EVEX.256.66.0F38.W0 46 /r VPSRAVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.
EVEX.512.66.0F38.W0 46 /r VPSRAVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.
EVEX.128.66.0F38.W1 46 /r VPSRAVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.
EVEX.256.66.0F38.W1 46 /r VPSRAVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.
EVEX.512.66.0F38.W1 46 /r VPSRAVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element is filled with the corresponding sign bit of the source element.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRAVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN
            COUNT := SRC2[i+3:i]
            IF COUNT < 16
                THEN    DEST[i+15:i] := SignExtend(SRC1[i+15:i] >> COUNT)
                ELSE
                    FOR k := 0 TO 15
                        DEST[i+k] := SRC1[i+15]
                    ENDFOR;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRAVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
```

```
   (* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] := SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] := 0;
```

VPSRAVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
   (* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
   (* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] := SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAXVL-1:256] := 0;
```

VPSRAVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
               COUNT := SRC2[4:0]
               IF COUNT < 32
                  THEN    DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
                  ELSE
                     FOR k := 0 TO 31
                        DEST[i+k] := SRC1[i+31]
                     ENDFOR;
               FI
            ELSE
               COUNT := SRC2[i+4:i]
               IF COUNT < 32
                  THEN    DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
                  ELSE
                     FOR k := 0 TO 31
                        DEST[i+k] := SRC1[i+31]
                     ENDFOR;
               FI
         FI;
   ELSE
      IF *merging-masking*                ; merging-masking
         THEN *DEST[31:0] remains unchanged*
         ELSE                             ; zeroing-masking
            DEST[31:0] := 0
         FI
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRAVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask* THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
```

```
                THEN
                    COUNT := SRC2[5:0]
                    IF COUNT < 64
                        THEN    DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 63
                                DEST[i+k] := SRC1[i+63]
                            ENDFOR;
                    FI
                ELSE
                    COUNT := SRC2[i+5:i]
                    IF COUNT < 64
                        THEN    DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k := 0 TO 63
                                DEST[i+k] := SRC1[i+63]
                            ENDFOR;
                    FI
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i _mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m512i _mm512_srav_epi16(__m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_mask_srav_epi16(__m512i s, __mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_maskz_srav_epi16(__mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m128i _mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVW __m128i _mm_mask_srav_epi16(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi32 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.128.66.0F38.W1 10 /r VPSRLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 10 /r VPSRLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 10 /r VPSRLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW OR AVX10.1	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W0 45 /r VPSRLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W0 45 /r VPSRLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W0 45 /r VPSRLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F OR AVX10.1	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.128.66.0F38.W1 45 /r VPSRLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.256.66.0F38.W1 45 /r VPSRLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.512.66.0F38.W1 45 /r VPSRLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F OR AVX10.1	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] >> SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRLVD (VEX.128 version)

```
COUNT_0 := SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
    DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] := ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
    DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSRLVD (VEX.256 version)

```
COUNT_0 := SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] := 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] := ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
    DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSRLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
                ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

VPSRLVQ (VEX.128 version)

```
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
    DEST[127:64] := ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
    DEST[127:64] := 0;
DEST[MAXVL-1:128] := 0;
```

VPSRLVQ (VEX.256 version)

```
COUNT_0 := SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] := 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] := ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
    DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;
```

VPSRLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
                ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16( __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i _mm256_maskz_srlv_epi16( __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m128i _mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m128i _mm_maskz_srlv_epi16( __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m256i _mm256_srlv_epi32 (__m256i m, __m256i count)
VPSRLVD __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_maskz_srlv_epi32( __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m256i _mm256_mask_srlv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m256i _mm256_maskz_srlv_epi32( __mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m128i _mm_mask_srlv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVD __m128i _mm_maskz_srlv_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_maskz_srlv_epi64( __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i _mm256_mask_srlv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m256i _mm256_maskz_srlv_epi64( __mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m128i _mm_mask_srlv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_maskz_srlv_epi64( __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m256i _mm256_srlv_epi64 (__m256i m, __m256i count)
VPSRLVD __m128i _mm_srlv_epi32( __m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_srlv_epi64( __m128i a, __m128i cnt);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPSRLVD/Q, see Table 2-51, “Type E4 Class Exception Conditions.”

EVEX-encoded VPSRLVW, see Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking xmm1, xmm2, and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking ymm1, ymm2, and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Bitwise ternary logic taking zmm1, zmm2, and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking xmm1, xmm2, and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise ternary logic taking ymm1, ymm2, and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Bitwise ternary logic taking zmm1, zmm2, and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second, and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location The destination operand is a ZMM register conditionally updated with writemask k1.

Table 5-20 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

Table 5-20. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values

VPTERNLOGD reg1, reg2, src3, 0xE2			Bit Result with Imm8=0xE2	VPTERNLOGD reg1, reg2, src3, 0xE4			Bit Result with Imm8=0xE4
Bit(reg1)	Bit(reg2)	Bit(src3)		Bit(reg1)	Bit(reg2)	Bit(src3)	
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

Operation

VPTERNLOGD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            FOR k := 0 TO 31
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]
                    ELSE DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]
                FI;
                        ; table lookup of immediate bellow;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31+i:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31+i:i] := 0
        FI;
    FI;
ENDFOR;
```

```
DEST[MAXVL-1:VL] := 0
```

VPTERNLOGQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            FOR k := 0 TO 63
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]
                    ELSE DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]
                FI;                     ; table lookup of immediate bellow;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[63+i:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[63+i:i] := 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPTERNLOGD __m512i _mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_maskz_ternarylogic_epi32( __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_maskz_ternarylogic_epi32( __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i _mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_maskz_ternarylogic_epi64( __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i _mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_mask_ternarylogic_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_maskz_ternarylogic_epi64( __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i _mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_mask_ternarylogic_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_maskz_ternarylogic_epi64( __mmask8 m, __m128i a, __m128i b, int imm);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW OR AVX10.1	Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the write-mask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

VPTESTMD/VPTESTMQ: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

VPTESTMB/VPTESTMW: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

Operation

VPTESTMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j * 8
    IF k1[j] OR *no writemask*
        THEN    DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                         ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN    DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                         ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
            FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTMQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE    DEST[j] := 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPTESTMB __mmask64 _mm512_test_epi8_mask( __m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask( __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask( __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask(__m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask(__mmask8, __m512i a, __m512i b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPTESTMD/Q: See Table 2-51, “Type E4 Class Exception Conditions.”

VPTESTMB/W: See Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VPTESTNMB/W/D/Q—Logical NAND and Set

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, zmm2, zmm3/m512	A	V/V	(AVX512F AND AVX512BW) OR AVX10.1	Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, xmm2, xmm3/m128	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, ymm2, ymm3/m256	A	V/V	(AVX512VL AND AVX512BW) OR AVX10.1	Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, zmm2, zmm3/m512	A	V/V	(AVX512F AND AVX512BW) OR AVX10.1	Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F OR AVX10.1	Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F OR AVX10.1	Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Operation

VPTESTNMB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
    i := j*8
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j*16
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0
                ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0
            FI
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

VPTESTNMQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] == 0)? 1 : 0;
                ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] == 0)? 1 : 0;
            FI;
        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPTESTNMB __mmask64 _mm512_testn_epi8_mask( __m512i a, __m512i b);
VPTESTNMB __mmask64 _mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 _mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 _mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 _mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 _mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 _mm512_testn_epi16_mask( __m512i a, __m512i b);
VPTESTNMW __mmask32 _mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 _mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMW __mmask16 _mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 _mm_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMW __mmask8 _mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMD __mmask16 _mm512_testn_epi32_mask( __m512i a, __m512i b);
VPTESTNMD __mmask16 _mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 _mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMD __mmask8 _mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 _mm_testn_epi32_mask(__m128i a, __m128i b);
VPTESTNMD __mmask8 _mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 _mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm_testn_epi64_mask(__m128i a, __m128i b);
```

VPTESTNMQ __mmask8 _mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPTESTNMD/VPTESTNMQ: See Table 2-51, “Type E4 Class Exception Conditions.”

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb in Table 2-51, “Type E4 Class Exception Conditions.”

VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 50 /r ib VRANGEPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate two RANGE operation output value from 2 pairs of double precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.256.66.0F3A.W1 50 /r ib VRANGEPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate four RANGE operation output value from 4pairs of double precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.512.66.0F3A.W1 50 /r ib VRANGEPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate eight RANGE operation output value from 8 pairs of double precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

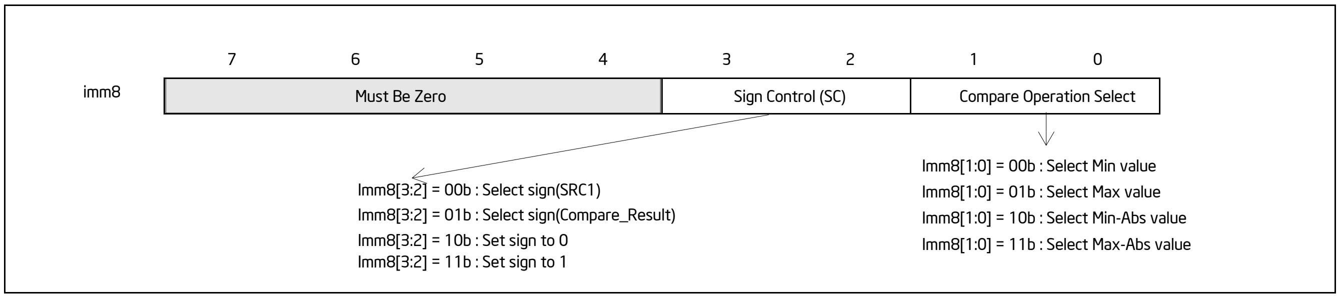

imm8
7
6
5
4
3
2
1
0
Must Be Zero
Sign Control (SC)
Compare Operation Select
Imm8[3:2] = 00b : Select sign(SRC1)
Imm8[3:2] = 01b : Select sign(Compare_Result)
Imm8[3:2] = 10b : Set sign to 0
Imm8[3:2] = 11b : Set sign to 1
Imm8[1:0] = 00b : Select Min value
Imm8[1:0] = 01b : Select Max value
Imm8[1:0] = 10b : Select Min-Abs value
Imm8[1:0] = 11b : Select Max-Abs value

Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Table 5-21. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]

Src1	Src2	Result	IE Signaling Due to Comparison	Imm8[3:2] Effect to Range Output
sNaN1	sNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	qNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	Norm2	Quiet(sNaN1)	Yes	Ignored
qNaN1	sNaN2	Quiet(sNaN2)	Yes	Ignored
qNaN1	qNaN2	qNaN1	No	Applicable
qNaN1	Norm2	Norm2	No	Applicable
Norm1	sNaN2	Quiet(sNaN2)	Yes	Ignored
Norm1	qNaN2	Norm1	No	Applicable

Table 5-22. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS, and MAX, MAX_ABS

MIN and MIN_ABS			MAX and MAX_ABS		
Src1	Src2	Result	Src1	Src2	Result
+0	-0	-0	+0	-0	+0
-0	+0	-0	-0	+0	+0

Table 5-23. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)

MIN_ABS (|a| = |b|, a>0, b<0)			MAX_ABS (|a| = |b|, a>0, b<0)		
Src1	Src2	Result	Src1	Src2	Result
a	b	b	a	b	a
b	a	b	b	a	a

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[62:52];
    Src1.fraction := SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp := SRC2[62:52];
    Src2.fraction := SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[63:0] := SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] := SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest := TMP[63:0];// Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}
```

```
CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:

zmm_dst is the destination operand.
zmm_src is the input operand to compare against ±1023 (this is SRC1).
zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e., SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGEPD __m512d _mm512_range_pd ( __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_range_round_pd ( __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_mask_range_pd (__m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd ( __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd ( __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd ( __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd ( __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm_range_pd ( __m128 a, __m128d b, int imm);
VRANGEPD __m128d _mm_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm_maskz_range_pd ( __mmask8 k, __m128d a, __m128d b, int imm);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

VRANGEPS—Range Restriction Calculation for Packed Pairs of Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[30:23];
    Src1.fraction := SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp := SRC2[30:23];
    Src2.fraction := SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[31:0] := SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] := SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest := (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest := TMP[31:0];// Preserve sign of compare result
    10: dest := (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest := (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPS

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;

Where:

zmm_dst is the destination operand.

zmm_src is the input operand to compare against ±150.

zmm_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |zmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGEPS __m512 _mm512_range_ps ( __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_range_round_ps ( __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_maskz_range_ps ( __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_maskz_range_round_ps ( __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 _mm256_range_ps ( __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_maskz_range_ps ( __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 _mm_range_ps ( __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_maskz_range_ps ( __mmask8 k, __m128 a, __m128 b, int imm);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRANGESD—Range Restriction Calculation From a Pair of Scalar Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 51 /r VRANGESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate a RANGE operation output value from 2 double precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

This instruction calculates a range operation output from two input double precision floating-point values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[62:52];
    Src1.fraction := SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp := SRC2[62:52];
    Src2.fraction := SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[63:0] := SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] := SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest := TMP[63:0];// Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGESD

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] := RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;

Where:

xmm_dst is the destination operand.

xmm_src is the input operand to compare against ±1023.

xmm_1023 is the reference operand, contains the value of 1023.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 1023, then its value will be written into xmm_dst. Otherwise, the value stored in xmm_dst will get the value of 1023 (received on xmm_1023).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if xmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023while if xmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGESD __m128d _mm_range_sd ( __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 51 /r VRANGESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction calculates a range operation output from two input single precision floating-point values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-21. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-21.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-22.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-23.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-21
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[30:23];
    Src1.fraction := SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp := SRC2[30:23];
    Src2.fraction := SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[31:0] := SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] := SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] := from Table 5-22
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-23
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest := (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest := TMP[31:0];// Preserve sign of compare result
    10: dest := (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest := (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGESS

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] := RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] = 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGESS zmm_dst, zmm_src, zmm_150, 02h;

Where:

xmm_dst is the destination operand.

xmm_src is the input operand to compare against ±150.

xmm_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in xmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if xmm_src < -150, the result of VRANGESS will be the minimal value of -150 while if xmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGESS __m128 _mm_range_ss ( __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4C /r VRCP14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4C /r VRCP14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4C /r VRCP14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocals of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand. The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-24. VRCP14PD/VRCP14SD Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \le X \le -0$	-INF	Very small denormal
$X > 2^{1022}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{1022}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14PD ((EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := APPROXIMATE(1.0/SRC[63:0]);
                ELSE DEST[i+63:i] := APPROXIMATE(1.0/SRC[i+63:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PD __m512d _mm512_rcp14_pd(__m512d a);

VRCP14PD __m512d _mm512_mask_rcp14_pd(__m512d s, __mmask8 k, __m512d a);

VRCP14PD __m512d _mm512_maskz_rcp14_pd(__mmask8 k, __m512d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4C /r VRCP14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4C /r VRCP14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4C /r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single precision floating-point values in the source operand (the second operand) and stores the packed single precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-25. VRCP14PS/VRCP14SS Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-128}$	INF	Very small denormal
$-2^{-128} \le X \le -0$	-INF	Very small denormal
$X > 2^{126}$	Underflow	Up to 18 bits of fractions are returned[1]
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned[1]
$X = 2^{-n}$	2^n	
$X = -2^{-n}$	-2^n	

NOTES:

1. In this case, the mantissa is shifted right by one or two bits.

A numerically exact implementation of VRCP14xx can be found at:

https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := APPROXIMATE(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] := APPROXIMATE(1.0/SRC[i+31:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRCP14PS __m512 _mm512_rcp14_ps( __m512 a);
VRCP14PS __m512 _mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 _mm512_maskz_rcp14_ps( __mmask16 k, __m512 a);
VRCP14PS __m256 _mm256_rcp14_ps( __m256 a);
VRCP14PS __m256 _mm512_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 _mm512_maskz_rcp14_ps( __mmask8 k, __m256 a);
VRCP14PS __m128 _mm_rcp14_ps( __m128 a);
VRCP14PS __m128 _mm_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 _mm_maskz_rcp14_ps( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, “Type E4 Class Exception Conditions.”

VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal of the scalar double precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-24 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at:

https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SD (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] := APPROXIMATE(1.0/SRC2[63:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SD __m128d _mm_rcp14_sd(__m128d a, __m128d b);

VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);

VRCP14SD __m128d _mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-53, “Type E5 Class Exception Conditions.”

VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-25 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SS (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] := APPROXIMATE(1.0/SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SS __m128 _mm_rcp14_ss(__m128 a, __m128 b);
VRCP14SS __m128 _mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128 _mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-53, “Type E5 Class Exception Conditions.”

VRCPPH—Compute Reciprocals of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 4C /r VRCPPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 4C /r VRCPPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 4C /r VRCPPH zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX512-FP16 OR AVX10.1	Compute the approximate reciprocals of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of 8/16/32 packed FP16 values in the source operand (the second operand) and stores the packed FP16 results in the destination operand. The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

For special cases, see Table 5-26.

Table 5-26. VRCPPH/VRCPSH Special Cases

Input Value	Result Value	Comments
$0 \le X \le 2^{-16}$	INF	Very small denormal
$-2^{-16} \le X \le -0$	–INF	Very small denormal
$X > +\infty$	+0	
$X < -\infty$	–0	
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

Operation

```
VRCPPH dest{k1}, src
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := APPROXIMATE(1.0 / tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRCPPH __m128h _mm_mask_rcp_ph (__m128h src, __mmask8 k, __m128h a);
VRCPPH __m128h _mm_maskz_rcp_ph (__mmask8 k, __m128h a);
VRCPPH __m128h _mm_rcp_ph (__m128h a);
VRCPPH __m256h _mm256_mask_rcp_ph (__m256h src, __mmask16 k, __m256h a);
VRCPPH __m256h _mm256_maskz_rcp_ph (__mmask16 k, __m256h a);
VRCPPH __m256h _mm256_rcp_ph (__m256h a);
VRCPPH __m512h _mm512_mask_rcp_ph (__m512h src, __mmask32 k, __m512h a);
VRCPPH __m512h _mm512_maskz_rcp_ph (__mmask32 k, __m512h a);
VRCPPH __m512h _mm512_rcp_ph (__m512h a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VRCPSH—Compute Reciprocal of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 4D /r VRCPSH xmm1{k1}{z}, xmm2, xmm3/m16	A	V/V	AVX512-FP16 OR AVX10.1	Compute the approximate reciprocal of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For special cases, see Table 5-26.

Operation

VRCPSH dest{k1}, src1, src2

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := APPROXIMATE(1.0 / src2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRCPSH __m128h _mm_mask_rcp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VRCPSH __m128h _mm_maskz_rcp_sh (__mmask8 k, __m128h a, __m128h b);
VRCPSH __m128h _mm_rcp_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-60, “Type E10 Class Exception Conditions.”

VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 56 /r ib VREDUCEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed double precision floating-point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W1 56 /r ib VREDUCEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed double precision floating-point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W1 56 /r ib VREDUCEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Perform reduction transformation on double precision floating-point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Perform reduction transformation of the packed binary encoded double precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where “Round()” treats “src”, “2^M”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

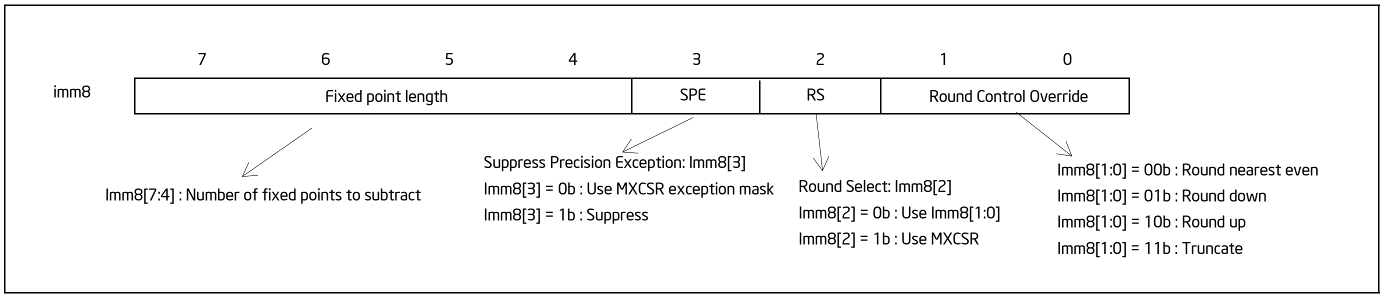

7
6
5
4
3
2
1
0
imm8
Fixed point length
SPE
RS
Round Control Override
Imm8[7:4] : Number of fixed points to subtract
Suppress Precision Exception: Imm8[3]
Imm8[3] = 0b : Use MXCSR exception mask
Imm8[3] = 1b : Suppress
Round Select: Imm8[2]
Imm8[2] = 0b : Use Imm8[1:0]
Imm8[2] = 1b : Use MXCSR
Imm8[1:0] = 00b : Round nearest even
Imm8[1:0] = 01b : Round down
Imm8[1:0] = 10b : Round up
Imm8[1:0] = 11b : Truncate

Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-27.

Table 5-27. VREDUCEPD/SD/PS/SS Special Cases

	Round Mode	Returned value
$\|Src1\| < 2^{-M-1}$	RNE	Src1
	RPI, Src1 > 0	Round $(Src1-2^{-M})$ *
	RPI, Src1 ≤ 0	Src1
	RNI, Src1 ≥ 0	Src1
$\|Src1\| < 2^{-M}$	RNI, Src1 < 0	Round $(Src1+2^{-M})$ *

Src1 = ±0, or
Dest = ±0 (Src1!=INF)
	NOT RNI	+0.0

	RNI	-0.0
Src1 = ±INF	any	+0.0
Src1= ±NAN	n/a	QNaN(Src1)

* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

Operation

```
ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[63:0] := $2^{-M}$ *{ROUND($2^{M}$*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and $2^{M}$ as standard binary FP values
    TMP[63:0] := SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCEPD

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask* THEN
           IF (EVEX.b == 1) AND (SRC *is memory*)
               THEN DEST[i+63:i] := ReduceArgumentDP(SRC[63:0], imm8[7:0]);
               ELSE DEST[i+63:i] := ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);
           FI;
   ELSE
       IF *merging-masking*                    ; merging-masking
           THEN *DEST[i+63:i] remains unchanged*
           ELSE                                ; zeroing-masking
               DEST[i+63:i] = 0
       FI;
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d a, int imm)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d _mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d _mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

Additionally:

#UD	If EVEX.vvvv != 1111B.

VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 56 /r /ib VREDUCEPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Perform reduction transformation on packed FP16 values in xmm2/m128/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1.
EVEX.256.NP.0F3A.W0 56 /r /ib VREDUCEPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Perform reduction transformation on packed FP16 values in ymm2/m256/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in ymm1 subject to writemask k1.
EVEX.512.NP.0F3A.W0 56 /r /ib VREDUCEPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Perform reduction transformation on packed FP16 values in zmm2/m512/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction performs a reduction transformation of the packed binary encoded FP16 values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4]. Specifically, the reduction transformation can be expressed as:

$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M}$

where ROUND() treats src, 2^M, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src = 2^p * man2, where 'man2' is the normalized significand and 'p' is the unbiased exponent.

Then if RC=RNE: $0 \leq |\text{ReducedResult}| \leq 2^{-M-1}$.

Then if RC ≠ RNE: $0 \leq |\text{ReducedResult}| < 2^{-M}$.

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-28.

Table 5-28. VREDUCEPH/VREDUCESH Special Cases

Input value	Round Mode	Returned Value
$\lvert Src1\rvert < 2^{-M-1}$	RNE	Src1
$\lvert Src1\rvert < 2^{-M}$	RU, Src1 > 0	$Round(Src1 - 2^{-M})$[1]
	RU, Src1 ≤ 0	Src1
	RD, Src1 ≥ 0	Src1
	RD, Src1 < 0	$Round(Src1 + 2^{-M})$
Src1 = ±0 or Dest = ±0 (Src1 ≠ ∞)	NOT RD	+0.0
	RD	−0.0
Src1 = ±∞	Any	+0.0
Src1 = ±NAN	Any	QNaN (Src1)

NOTES:

1. The Round(.) function uses rounding controls specified by (imm8[2]? MXCSR.RC: imm8[1:0]).

Operation

```
def reduce_fp16(src, imm8):
    nan := (src.exp = 0x1F) and (src.fraction != 0)
    if nan:
        return QNAN(src)
    m := imm8[7:4]
    rc := imm8[1:0]
    rc_source := imm8[2]
    spe := imm[3] // suppress precision exception
    tmp := 2^(-m) * ROUND(2^m * src, spe, rc_source, rc)
    tmp := src - tmp // using same RC, SPE controls
    return tmp
```

VREDUCEPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := reduce_fp16(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPH __m128h _mm_mask_reduce_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_maskz_reduce_ph (__mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_reduce_ph (__m128h a, int imm8);
VREDUCEPH __m256h _mm256_mask_reduce_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_maskz_reduce_ph (__mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_reduce_ph (__m256h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_maskz_reduce_ph (__mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_reduce_ph (__m512h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_maskz_reduce_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_reduce_round_ph (__m512h a, int imm8, const int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 56 /r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed single-precision floating-point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W0 56 /r ib VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Perform reduction transformation on packed single-precision floating-point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W0 56 /r ib VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Perform reduction transformation on packed single-precision floating-point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Perform reduction transformation of the packed binary encoded single precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where “Round()” treats “src”, “2^M”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,

where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-27.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[31:0] := 2^-M *{ROUND(2^M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] := SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCEPS

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b == 1) AND (SRC *is memory*)
                    THEN DEST[i+31:i] := ReduceArgumentSP(SRC[31:0], imm8[7:0]);
                    ELSE DEST[i+31:i] := ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
                FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VREDUCEPS __m512 _mm512_mask_reduce_ps( __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 _mm256_mask_reduce_ps( __m256 a, int imm)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 _mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps( __m128 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 _mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions”; additionally:

#UD	If EVEX.vvvv != 1111B.

VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 57 VREDUCESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8/r	A	V/V	AVX512DQ OR AVX10.1	Perform a reduction transformation on a scalar double precision floating-point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Perform a reduction transformation of the binary encoded double precision floating-point value in the low qword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^{M}*src))*2^{-M};

where “Round()” treats “src”, “2^{M}”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^{p}*man2,

where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-27.

Operation

```
ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[63:0] := $2^{-M}$ *{ROUND($2^M$*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and $2^M$ as standard binary FP values
    TMP[63:0] := SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESD

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := ReduceArgumentDP(SRC2[63:0], imm8[7:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VREDUCESD __m128d _mm_mask_reduce_sd( __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 57 /r /ib VREDUCESH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Perform a reduction transformation on the low binary encoded FP16 value in xmm3/m16 by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction performs a reduction transformation of the low binary encoded FP16 value in the source operand (the second operand) and store the reduced result in binary FP format to the low element of the destination operand (the first operand) under the writemask k1. For further details see the description of VREDUCEPH.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-28.

Operation

VREDUCESH dest{k1}, src, imm8

```
IF k1[0] or *no writemask*:
    dest.fp16[0] := reduce_fp16(src2.fp16[0], imm8)     // see VREDUCEPH
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else dest.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VREDUCESH __m128h _mm_mask_reduce_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_maskz_reduce_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_reduce_round_sh (__m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_mask_reduce_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_maskz_reduce_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_reduce_sh (__m128h a, __m128h b, int imm8);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 57 /r /ib VREDUCESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ OR AVX10.1	Perform a reduction transformation on a scalar single-precision floating-point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Perform a reduction transformation of the binary encoded single precision floating-point value in the low dword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^{M}*src))*2^{-M};

where “Round()” treats “src”, “2^{M}”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^{p}*man2,

where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-27.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC := imm8[1:0];// Round Control for ROUND() operation
    RC source := imm[2];
    SPE := imm[3];// Suppress Precision Exception
    TMP[31:0] := $2^{-M}$ *{ROUND($2^M$*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and $2^M$ as standard binary FP values
    TMP[31:0] := SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESS

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := ReduceArgumentSP(SRC2[31:0], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] = 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VREDUCESS __m128 _mm_mask_reduce_ss( __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, "Type E3 Class Exception Conditions."

VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 09 /r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed double precision floating-point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W1 09 /r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed double precision floating-point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W1 09 /r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds packed double precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the double precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

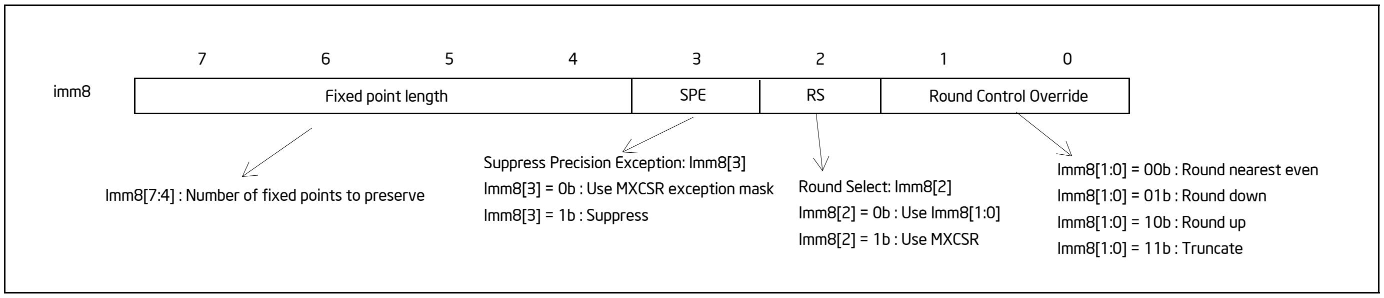

7
6
5
4
3
2
1
0
imm8
Fixed point length
SPE
RS
Round Control Override
Imm8[7:4] : Number of fixed points to preserve
Suppress Precision Exception: Imm8[3]
Imm8[3] = 0b : Use MXCSR exception mask
Imm8[3] = 1b : Suppress
Round Select: Imm8[2]
Imm8[2] = 0b : Use Imm8[1:0]
Imm8[2] = 1b : Use MXCSR
Imm8[1:0] = 00b : Round nearest even
Imm8[1:0] = 01b : Round down
Imm8[1:0] = 10b : Round up
Imm8[1:0] = 11b : Truncate

Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-29.

Table 5-29. VRNDSCALEPD/SD/PS/SS Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

Operation

```
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC        ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]       ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]          ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] := round_to_nearest_even_integer($2^M$*SRC[63:0])
    01: TMP[63:0] := round_to_equal_or_smaller_integer($2^M$*SRC[63:0])
    10: TMP[63:0] := round_to_equal_or_larger_integer($2^M$*SRC[63:0])
    11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer($2^M$*SRC[63:0])
    ESAC

    Dest[63:0] := $2^{-M}$* TMP[63:0]          ; scale down back to $2^{-M}$

    if (imm8[3] = 0) Then   ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then       ; check precision lost
            set_precision()                     ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}
```

VRNDSCALEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF *src is a memory operand*
    THEN TMP_SRC := BROADCAST64(SRC, VL, k1)
    ELSE TMP_SRC := SRC
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALEPD __m512d _mm512_roundscale_pd( __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd( __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd( __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd( __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd( __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd( __m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd( __mmask8 k, __m128d a, int imm);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, "Type E2 Class Exception Conditions."

VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.0F3A.W0 08 /r /ib VRNDSCALEPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Round packed FP16 values in xmm2/m128/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1.
EVEX.256.NP.0F3A.W0 08 /r /ib VRNDSCALEPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Round packed FP16 values in ymm2/m256/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in ymm1 subject to writemask k1.
EVEX.512.NP.0F3A.W0 08 /r /ib VRNDSCALEPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Round packed FP16 values in zmm2/m512/m16bcst to a number of fraction bits specified by the imm8 field. Store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8 (r)	N/A

Description

This instruction rounds the FP16 values in the source operand by the rounding mode specified in the immediate operand (see Table 5-30) and places the result in the destination operand. The destination operand is conditionally updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as an FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 5-30, "Imm8 Controls for VRNDSCALEPH/VRNDSCALESH." Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-31.

The formula of the operation on each data element for VRNDSCALEPH is

ROUND(x) = 2^{-M} *Round_to_INT(x * 2^M, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x * 2^M is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

If this instruction encoding's SPE bit (bit 3) in the immediate operand is 1, VRNDSCALEPH can set MXCSR.UE without MXCSR.PE.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Table 5-30. Imm8 Controls for VRNDSCALEPH/VRNDSCALESH

Imm8 Bits	Description
imm8[7:4]	Number of fixed points to preserve.
imm8[3]	Suppress Precision Exception (SPE) 0b00: Implies use of MXCSR exception mask. 0b01: Implies suppress.
imm8[2]	Round Select (RS) 0b00: Implies use of imm8[1:0]. 0b01: Implies use of MXCSR.
imm8[1:0]	Round Control Override: 0b00: Round nearest even. 0b01: Round down. 0b10: Round up. 0b11: Truncate.

Table 5-31. VRNDSCALEPH/VRNDSCALESH Special Cases

Input Value	Returned Value
Src1 = ±∞	Src1
Src1 = ±NaN	Src1 converted to QNaN
Src1 = ±0	Src1

Operation

```
def round_fp16_to_integer(src, imm8):
    if imm8[2] = 1:
        rounding_direction := MXCSR.RC
    else:
        rounding_direction := imm8[1:0]
    m := imm8[7:4] // scaling factor

    tsrc1 := 2^m * src

    if rounding_direction = 0b00:
        tmp := round_to_nearest_even_integer(trc1)
    else if rounding_direction = 0b01:
        tmp := round_to_equal_or_smaller_integer(trc1)
    else if rounding_direction = 0b10:
        tmp := round_to_equal_or_larger_integer(trc1)
    else if rounding_direction = 0b11:
        tmp := round_to_smallest_magnitude_integer(trc1)

    dst := 2^(-m) * tmp

    if imm8[3]==0: // check SPE
        if src != dst:
            MXCSR.PE := 1
    return dst
```

VRNDSCALEPH dest{k1}, src, imm8

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := round_fp16_to_integer(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALEPH __m128h _mm_mask_roundscale_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VRNDSCALEPH __m128h _mm_maskz_roundscale_ph (__mmask8 k, __m128h a, int imm8);
VRNDSCALEPH __m128h _mm_roundscale_ph (__m128h a, int imm8);
VRNDSCALEPH __m256h _mm256_mask_roundscale_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VRNDSCALEPH __m256h _mm256_maskz_roundscale_ph (__mmask16 k, __m256h a, int imm8);
VRNDSCALEPH __m256h _mm256_roundscale_ph (__m256h a, int imm8);
VRNDSCALEPH __m512h _mm512_mask_roundscale_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_maskz_roundscale_ph (__mmask32 k, __m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_roundscale_ph (__m512h a, int imm8);
VRNDSCALEPH __m512h _mm512_mask_roundscale_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VRNDSCALEPH __m512h _mm512_maskz_roundscale_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VRNDSCALEPH __m512h _mm512_roundscale_round_ph (__m512h a, int imm8, const int sae);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 08 /r ib VRNDSCALEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed single-precision floating-point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W0 08 /r ib VRNDSCALEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Rounds packed single-precision floating-point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W0 08 /r ib VRNDSCALEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

Description

Round the single precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

$\text{ROUND}(x) = 2^{-M}\text{*Round\_to\_INT}(x\text{*}2^{M}, \text{round\_ctrl}),$

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of $x\text{*}2^{M}$ is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-29.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC          ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]         ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]              ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] := round_to_nearest_even_integer($2^M$*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer($2^M$*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer($2^M$*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer($2^M$*SRC[31:0])
    ESAC;

    Dest[31:0] := $2^{-M}$* TMP[31:0]             ; scale down back to $2^{-M}$
    if (imm8[3] = 0) Then           ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then       ; check precision lost
            set_precision()             ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}
```

VRNDSCALEPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF *src is a memory operand*
    THEN TMP_SRC := BROADCAST32(SRC, VL, k1)
    ELSE TMP_SRC := SRC
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := RoundToIntegerSP(TMP_SRC[i+31:i]), imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALEPS __m512 _mm512_roundscale_ps( __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_roundscale_round_ps( __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_ps( __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_round_ps( __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 _mm256_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_maskz_roundscale_ps( __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 _mm_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m128 _mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 _mm_maskz_roundscale_ps( __mmask8 k, __m128 a, int imm);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

VRNDSCALESD—Round Scalar Float64 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds scalar double precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Rounds a double precision floating-point value in the low quadword (see Figure 5-29) element of the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESD is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-29.

Operation

```
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC        ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]       ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]            ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] := round_to_nearest_even_integer($2^{M}$*SRC[63:0])
    01: TMP[63:0] := round_to_equal_or_smaller_integer($2^{M}$*SRC[63:0])
    10: TMP[63:0] := round_to_equal_or_larger_integer($2^{M}$*SRC[63:0])
    11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer($2^{M}$*SRC[63:0])
    ESAC

    Dest[63:0] := $2^{-M}$* TMP[63:0]         ; scale down back to $2^{-M}$

    if (imm8[3] = 0) Then    ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then        ; check precision lost
            set_precision()                  ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}
```

VRNDSCALESD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                  ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                              ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALESD __m128d _mm_roundscale_sd ( __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_roundscale_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_mask_roundscale_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VRNDSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.0F3A.W0 0A /r /ib VRNDSCALESH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX512-FP16 OR AVX10.1	Round the low FP16 value in xmm3/m16 to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

This instruction rounds the low FP16 value in the second source operand by the rounding mode specified in the immediate operand (see Table 5-30) and places the result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as a FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 5-30, “Imm8 Controls for VRNDSCALEPH/VRNDSCALESH.” Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-31.

If this instruction encoding’s SPE bit (bit 3) in the immediate operand is 1, VRNDSCALESH can set MXCSR.UE without MXCSR.PE.

The formula of the operation on each data element for VRNDSCALESH is:

ROUND(x) = 2^{-M} *Round_to_INT(x * 2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x * 2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

Operation

VRNDSCALESH dest{k1}, src1, src2, imm8

```
IF k1[0] or *no writemask*:
    DEST.fp16[0] := round_fp16_to_integer(src2.fp16[0], imm8) // see VRNDSCALEPH
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] = src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALESH __m128h _mm_mask_roundscale_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int
                sae);
VRNDSCALESH __m128h _mm_maskz_roundscale_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VRNDSCALESH __m128h _mm_roundscale_round_sh (__m128h a, __m128h b, int imm8, const int sae);
VRNDSCALESH __m128h _mm_mask_roundscale_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);
VRNDSCALESH __m128h _mm_maskz_roundscale_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
VRNDSCALESH __m128h _mm_roundscale_sh (__m128h a, __m128h b, int imm8);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F OR AVX10.1	Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Rounds the single precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-29.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC          ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0]         ; get round control from imm8[1:0]
    FI
    M := imm8[7:4]              ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] := round_to_nearest_even_integer($2^M$*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer($2^M$*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer($2^M$*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer($2^M$*SRC[31:0])
    ESAC;

    Dest[31:0] := $2^{-M}$* TMP[31:0]           ; scale down back to $2^{-M}$
    if (imm8[3] = 0) Then           ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then       ; check precision lost
            set_precision()             ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}
```

VRNDSCALESS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRNDSCALESS __m128 _mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square roots of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC *is memory*)
                    THEN DEST[i+63:i] := APPROXIMATE(1.0/ SQRT(SRC[63:0]));
                    ELSE DEST[i+63:i] := APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));
                FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Table 5-32. VRSQRT14PD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRT14PD __m512d _mm512_rsqrt14_pd( __m512d a);
VRSQRT14PD __m512d _mm512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d _mm512_maskz_rsqrt14_pd( __mmask8 k, __m512d a);
VRSQRT14PD __m256d _mm256_rsqrt14_pd( __m256d a);
VRSQRT14PD __m256d _mm512_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d _mm512_maskz_rsqrt14_pd( __mmask8 k, __m256d a);
VRSQRT14PD __m128d _mm_rsqrt14_pd( __m128d a);
VRSQRT14PD __m128d _mm_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d _mm_maskz_rsqrt14_pd( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E4 Class Exception Conditions."

VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4E /r VRSQRT14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4E /r VRSQRT14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4E /r VRSQRT14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single precision floating-point values in the source operand (the second operand) and stores the packed single precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := APPROXIMATE(1.0/ SQRT(SRC[31:0]));
                ELSE DEST[i+31:i] := APPROXIMATE(1.0/ SQRT(SRC[i+31:i]));
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

Table 5-33. VRSQRT14PS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRT14PS __m512 _mm512_rsqrt14_ps( __m512 a);
VRSQRT14PS __m512 _mm512_mask_rsqrt14_ps(__m512 s, __mmask16 k, __m512 a);
VRSQRT14PS __m512 _mm512_maskz_rsqrt14_ps( __mmask16 k, __m512 a);
VRSQRT14PS __m256 _mm256_rsqrt14_ps( __m256 a);
VRSQRT14PS __m256 _mm256_mask_rsqrt14_ps(__m256 s, __mmask8 k, __m256 a);
VRSQRT14PS __m256 _mm256_maskz_rsqrt14_ps( __mmask8 k, __m256 a);
VRSQRT14PS __m128 _mm_rsqrt14_ps( __m128 a);
VRSQRT14PS __m128 _mm_mask_rsqrt14_ps(__m128 s, __mmask8 k, __m128 a);
VRSQRT14PS __m128 _mm_maskz_rsqrt14_ps( __mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square root of the scalar double precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the approximate reciprocal of the square roots of the scalar double precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14SD (EVEX version)

```
IF k1[0] or *no writemask*
   THEN      DEST[63:0] := APPROXIMATE(1.0/ SQRT(SRC2[63:0]))
   ELSE
       IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] := 0
       FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Table 5-34. VRSQRT14SD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SD __m128d _mm_rsqrt14_sd(__m128d a, __m128d b);
VRSQRT14SD __m128d _mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d _mm_maskz_rsqrt14_sd(__mmask8d m, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-53, “Type E5 Class Exception Conditions.”

VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F OR AVX10.1	Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes of the approximate reciprocal of the square root of the scalar single precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an ∞, zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14SS (EVEX version)

```
IF k1[0] or *no writemask*
   THEN    DEST[31:0] := APPROXIMATE(1.0/ SQRT(SRC2[31:0]))
   ELSE
       IF *merging-masking*                ; merging-masking
           THEN *DEST[31:0] remains unchanged*
           ELSE                            ; zeroing-masking
               THEN DEST[31:0] := 0
       FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Table 5-35. VRSQRT14SS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRT14SS __m128 _mm_rsqrt14_ss( __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_maskz_rsqrt14_ss( __mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-53, “Type E5 Class Exception Conditions.”

VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 4E /r VRSQRTPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of the square roots of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 4E /r VRSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compute the approximate reciprocals of the square roots of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 4E /r VRSQRTPH zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX512-FP16 OR AVX10.1	Compute the approximate reciprocals of the square roots of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a SIMD computation of the approximate reciprocals square-root of 8/16/32 packed FP16 floating-point values in the source operand (the second operand) and stores the packed FP16 floating-point results in the destination operand.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$. For special cases, see Table 5-36.

The destination elements are updated according to the writemask.

Table 5-36. VRSQRTPH/VRSQRTSH Special Cases

Input value	Reset Value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including –∞
X = –0	–∞	
X = +0	+∞	
X = +∞	+0	

Operation

VRSQRTPH dest{k1}, src

```
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := APPROXIMATE(1.0 / SQRT(tsrc) )
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRTPH __m128h _mm_mask_rsqrt_ph (__m128h src, __mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_maskz_rsqrt_ph (__mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_rsqrt_ph (__m128h a);
VRSQRTPH __m256h _mm256_mask_rsqrt_ph (__m256h src, __mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_maskz_rsqrt_ph (__mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_rsqrt_ph (__m256h a);
VRSQRTPH __m512h _mm512_mask_rsqrt_ph (__m512h src, __mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_maskz_rsqrt_ph (__mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_rsqrt_ph (__m512h a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-51, “Type E4 Class Exception Conditions.”

VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 4F /r VRSQRTSH xmm1{k1}{z}, xmm2, xmm3/m16	A	V/V	AVX512-FP16 OR AVX10.1	Compute the approximate reciprocal square root of the FP16 value in xmm3/m16 and store the result in the low word element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the computation of the approximate reciprocal square-root of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL−1:128 of the destination operand are zeroed.

For special cases, see Table 5-36.

Operation

```
VRSQRTSH dest{k1}, src1, src2
VL = 128, 256 or 512
KL := VL/16

IF k1[0] or *no writemask*:
    DEST.fp16[0] := APPROXIMATE(1.0 / SQRT(src2.fp16[0]))
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged
DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRTSH __m128h _mm_mask_rsqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_maskz_rsqrt_sh (__mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_rsqrt_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-60, “Type E10 Class Exception Conditions.”

VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 2C /r VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed double precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.
EVEX.256.66.0F38.W1 2C /r VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed double precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.
EVEX.512.66.0F38.W1 2C /r VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	A	V/V	AVX512F OR AVX10.1	Scale the packed double precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the packed double precision floating-point values in the first source operand by multiplying them by 2 to the power of the double precision floating-point values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{\text{floor(zmm3)}}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-38.

Table 5-37. VSCALEFPD/SD/PS/SS Special Cases

		Src2				Set IE
		±NaN	**+Inf**	**-Inf**	**0/Denorm/Norm**	
Src1	±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNAN
	±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
	±Inf	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNAN or -INF
	±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNAN or +INF
	Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNAN

Table 5-38. Additional VSCALEFPD/SD Special Cases

Special Case	Returned value	Faults
$\lvert result\rvert < 2^{-1074}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\lvert result\rvert \geq 2^{1024}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[63:0]);
                ELSE DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[i+63:i]);
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VSCALEFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m256d _mm256_scalef_pd(__m256d a, __m256d b);
VSCALEFPD __m256d _mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m128d _mm_scalef_pd(__m128d a, __m128d b);
VSCALEFPD __m128d _mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VSCALEFPD __m128d _mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

VSCALEFPH—Scale Packed FP16 Values with FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.MAP6.W0 2C /r VSCALEFPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Scale the packed FP16 values in xmm2 using values from xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 2C /r VSCALEFPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Scale the packed FP16 values in ymm2 using values from ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 2C /r VSCALEFPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Scale the packed FP16 values in zmm2 using values from zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a floating-point scale of the packed FP16 values in the first source operand by multiplying it by 2 to the power of the FP16 values in second source operand. The destination elements are updated according to the writemask.

The equation of this operation is given by:

$$zmm1 := zmm2 * 2^{floor(zmm3)}.$$

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits), and on the SAE bit.

Handling of special-case input values are listed in Table 5-39 and Table 5-40.

Table 5-39. VSCALEFPH/VSCALEFSH Special Cases

Src1	Src2				Set IE
	±NaN	+INF	−INF	0/Denorm/Norm	
±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNaN
±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
±INF	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNaN or −INF
±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNaN or +INF
Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNaN

Table 5-40. Additional VSCALEFPH/VSCALEFSH Special Cases

Special Case	Returned Value	Faults
$\lvert result\rvert < 2^{-24}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\lvert result\rvert \geq 2^{16}$	±INF (Src1 sign) or ±Max-Denormal (Src1 sign)	Overflow

Operation

```
def scale_fp16(src1,src2):
    tmp1 := src1
    tmp2 := src2
    return tmp1 * POW(2, FLOOR(tmp2))
```

VSCALEFPH dest{k1}, src1, src2

```
VL = 128, 256, or 512
KL := VL / 16

IF (VL = 512) AND (EVEX.b = 1) and no memory operand:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and (EVEX.b = 1):
            tsrc := src2.fp16[0]
        ELSE:
            tsrc := src2.fp16[i]
        dest.fp16[i] := scale_fp16(src1.fp16[i],tsrc)
    ELSE IF *zeroing*:
        dest.fp16[i] := 0
    //else dest.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPH __m128h _mm_mask_scalef_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSCALEFPH __m128h _mm_maskz_scalef_ph (__mmask8 k, __m128h a, __m128h b);
VSCALEFPH __m128h _mm_scalef_ph (__m128h a, __m128h b);
VSCALEFPH __m256h _mm256_mask_scalef_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VSCALEFPH __m256h _mm256_maskz_scalef_ph (__mmask16 k, __m256h a, __m256h b);
VSCALEFPH __m256h _mm256_scalef_ph (__m256h a, __m256h b);
VSCALEFPH __m512h _mm512_mask_scalef_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VSCALEFPH __m512h _mm512_maskz_scalef_ph (__mmask32 k, __m512h a, __m512h b);
VSCALEFPH __m512h _mm512_scalef_ph (__m512h a, __m512h b);
VSCALEFPH __m512h _mm512_mask_scalef_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, const int rounding);
VSCALEFPH __m512h _mm512_maskz_scalef_round_ph (__mmask32 k, __m512h a, __m512h b, const int;
VSCALEFPH __m512h _mm512_scalef_round_ph (__m512h a, __m512h b, const int rounding);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions”.

VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.
EVEX.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Scale the packed single-precision values in ymm2 using floating-point values from ymm3/m256/m32bcst. Under writemask k1.
EVEX.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	A	V/V	AVX512F OR AVX10.1	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the packed single precision floating-point values in the first source operand by multiplying them by 2 to the power of the float32 values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{\text{floor(zmm3)}}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-41.

Table 5-41. Additional VSCALEFPS/SS Special Cases

Special Case	Returned value	Faults
$\lvert result \rvert < 2^{-149}$	±0 or ±Min-Denormal (Src1 sign)	Underflow
$\lvert result \rvert \geq 2^{128}$	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{                ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[31:0]);
                    ELSE DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[i+31:i]);
                FI;
        ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_scalef_ps(__m512 a, __m512 b);
VSCALEFPS __m512 _mm512_mask_scalef_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m512 _mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m256 _mm256_scalef_ps(__m256 a, __m256 b);
VSCALEFPS __m256 _mm256_mask_scalef_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m256 _mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m128 _mm_scalef_ps(__m128 a, __m128 b);
VSCALEFPS __m128 _mm_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSCALEFPS __m128 _mm_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-48, “Type E2 Class Exception Conditions.”

VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	A	V/V	AVX512F OR AVX10.1	Scale the scalar double precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the scalar double precision floating-point value in the first source operand by multiplying it by 2 to the power of the double precision floating-point value in second source operand.

The equation of this operation is given by:

xmm1 := $\text{xmm2}*2^{\text{floor(xmm3)}}$.

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-38.

Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFSD (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] := SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VSCALEFSH—Scale Scalar FP16 Values with FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.MAP6.W0 2D /r VSCALEFSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Scale the FP16 values in xmm2 using the value from xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a floating-point scale of the low FP16 element in the first source operand by multiplying it by 2 to the power of the low FP16 element in second source operand, storing the result in the low element of the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The equation of this operation is given by:

$$xmm1 := xmm2 * 2^{floor(xmm3)}.$$

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

Handling of special-case input values are listed in Table 5-39 and Table 5-40.

Operation

VSCALEFSH dest{k1}, src1, src2

```
IF (EVEX.b = 1) and no memory operand:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] or *no writemask*:
    dest.fp16[0] := scale_fp16(src1.fp16[0], src2.fp16[0]) // see VSCALEFPH
ELSE IF *zeroing*:
    dest.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSH __m128h _mm_mask_scalef_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_maskz_scalef_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_scalef_round_sh (__m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_mask_scalef_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_maskz_scalef_sh (__mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_scalef_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.

VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	A	V/V	AVX512F OR AVX10.1	Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a floating-point scale of the scalar single precision floating-point value in the first source operand by multiplying it by 2 to the power of the float32 value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

Floor(xmm3) means maximum integer value $\leq$ xmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-37 and Table 5-41.

Operation

```
SCALE(SRC1, SRC2)
{
            ; Check for denormal operands
TMP_SRC2 := SRC2
TMP_SRC1 := SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFSS (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] := SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] := 0
        FI
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

Other Exceptions

See Table 2-49, “Type E3 Class Exception Conditions.”

VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single Precision, Packed Double Precision Floating-Point Values with Signed Dword and Qword Indices

Opcode/ Instruction	Op/E n	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, ymm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1	A	V/V	AVX512F OR AVX10.1	Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	N/A	N/A

Description

Stores up to four, eight, or 16 single precision elements (or two, four, or eight double precision elements) in doubleword/quadword vector xmm1, ymm1, or zmm1, to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the source register xmm, ymm, or zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be scattered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be scattered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

Operation

```
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement
```

VSCATTERDPS (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] :=
            SRC[i+31:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERDPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] :=
            SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERQPS (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] :=
            SRC[i+31:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

VSCATTERQPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] :=
            SRC[i+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERDPD void _mm512_i32scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERDPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m512 a, int scale);
VSCATTERDPD void _mm256_i32scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERDPD void _mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERDPS void _mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void _mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void _mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERQPS void _mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERDPD void _mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void _mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void _mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void _mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void _mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void _mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "Type E12 Class Exception Conditions."

VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-Bit Granularity

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed double precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed double precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.
EVEX.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F OR AVX10.1	Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

256-bit Version: Moves one of the two 128-bit packed single precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

Operation

```
Select2(SRC, control) {
CASE (control[0]) OF
    0:    TMP := SRC[127:0];
    1:    TMP := SRC[255:128];
ESAC;
RETURN TMP
}

Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP := SRC[127:0];
    1:    TMP := SRC[255:128];
    2:    TMP := SRC[383:256];
    3:    TMP := SRC[511:384];
ESAC;
RETURN TMP
}
```

VSHUFF32x4 (EVEX versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+31:i] := 0
            FI;
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFF64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFI32x4 (EVEX 512-bit version)

```
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
```

```
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VSHUFI64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFI32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512 _mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i _mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i b, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);
VSHUFF64x2 __m512d _mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-52, “Type E4NF Class Exception Conditions.”

Additionally:

#UD	If EVEX.L’L = 0 for VSHUFF32x4/VSHUFF64x2.

VSQRTPH—Compute Square Root of Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 51 /r VSQRTPH xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compute square roots of the packed FP16 values in xmm2/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 51 /r VSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Compute square roots of the packed FP16 values in ymm2/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 51 /r VSQRTPH zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1	Compute square roots of the packed FP16 values in zmm2/m512/m16bcst, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction performs a packed FP16 square-root computation on the values from source operand and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

```
VSQRTPH dest{k1}, src
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := SQRT(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPH __m128h _mm_mask_sqrt_ph (__m128h src, __mmask8 k, __m128h a);
VSQRTPH __m128h _mm_maskz_sqrt_ph (__mmask8 k, __m128h a);
VSQRTPH __m128h _mm_sqrt_ph (__m128h a);
VSQRTPH __m256h _mm256_mask_sqrt_ph (__m256h src, __mmask16 k, __m256h a);
VSQRTPH __m256h _mm256_maskz_sqrt_ph (__mmask16 k, __m256h a);
VSQRTPH __m256h _mm256_sqrt_ph (__m256h a);
VSQRTPH __m512h _mm512_mask_sqrt_ph (__m512h src, __mmask32 k, __m512h a);
VSQRTPH __m512h _mm512_maskz_sqrt_ph (__mmask32 k, __m512h a);
VSQRTPH __m512h _mm512_sqrt_ph (__m512h a);
VSQRTPH __m512h _mm512_mask_sqrt_round_ph (__m512h src, __mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h _mm512_maskz_sqrt_round_ph (__mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h _mm512_sqrt_round_ph (__m512h a, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

VSQRTSH—Compute Square Root of Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 51 /r VSQRTSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Compute square root of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a scalar FP16 square-root computation on the source operand and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

```
VSQRTSH dest{k1}, src1, src2
IF k1[0] or *no writemask*:
    DEST.fp16[0] := SQRT(src2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSH __m128h _mm_mask_sqrt_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_maskz_sqrt_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_sqrt_round_sh (__m128h a, __m128h b, const int rounding);
VSQRTSH __m128h _mm_mask_sqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSQRTSH __m128h _mm_maskz_sqrt_sh (__mmask8 k, __m128h a, __m128h b);
VSQRTSH __m128h _mm_sqrt_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-49, “Type E3 Class Exception Conditions.”

VSUBPH—Subtract Packed FP16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.NP.MAP5.W0 5C /r VSUBPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1	Subtract packed FP16 values from xmm3/m128/m16bcst to xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 5C /r VSUBPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	(AVX512-FP16 AND AVX512VL) OR AVX10.1[1]	Subtract packed FP16 values from ymm3/m256/m16bcst to ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 5C /r VSUBPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16 OR AVX10.1[1]	Subtract packed FP16 values from zmm3/m512/m16bcst to zmm2, and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction subtracts packed FP16 values from second source operand from the corresponding elements in the first source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VSUBPH (EVEX encoded versions) when src2 operand is a register

```
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

VSUBPH (EVEX encoded versions) when src2 operand is a memory source

```
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBPH __m128h _mm_mask_sub_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_maskz_sub_ph (__mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_sub_ph (__m128h a, __m128h b);
VSUBPH __m256h _mm256_mask_sub_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_maskz_sub_ph (__mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_sub_ph (__m256h a, __m256h b);
VSUBPH __m512h _mm512_mask_sub_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_maskz_sub_ph (__mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_sub_ph (__m512h a, __m512h b);
VSUBPH __m512h _mm512_mask_sub_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_maskz_sub_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_sub_round_ph (__m512h a, __m512h b, int rounding);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instruction, see Table 2-48, “Type E2 Class Exception Conditions.”

VSUBSH—Subtract Scalar FP16 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F3.MAP5.W0 5C /r VSUBSH xmm1{k1}{z}, xmm2, xmm3/m16 {er}	A	V/V	AVX512-FP16 OR AVX10.1	Subtract the low FP16 value in xmm3/m16 from xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction subtracts the low FP16 value from the second source operand from the corresponding value in the first source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

VSUBSH (EVEX encoded versions)

```
IF EVEX.b = 1 and SRC2 is a register:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := SRC1.fp16[0] - SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged
DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSH __m128h _mm_mask_sub_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_maskz_sub_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_sub_round_sh (__m128h a, __m128h b, int rounding);
VSUBSH __m128h _mm_mask_sub_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSUBSH __m128h _mm_maskz_sub_sh (__mmask8 k, __m128h a, __m128h b);
VSUBSH __m128h _mm_sub_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-49, "Type E3 Class Exception Conditions."

VUCOMISH—Unordered Compare Scalar FP16 Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.NP.MAP5.W0 2E /r VUCOMISH xmm1, xmm2/m16 {sae}	A	V/V	AVX512-FP16 OR AVX10.1	Compare low FP16 values in xmm1 and xmm2/m16 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VUCOMISH instruction differs from the VCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Operation

VUCOMISH

```
RESULT := UnorderedCompare(SRC1.fp16[0],SRC2.fp16[0])
if RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
else if RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
else if RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
else: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUCOMISH int _mm_ucomieq_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomige_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomigt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomile_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomilt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomineq_sh (__m128h a, __m128h b);
```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-50, "Type E3NF Class Exception Conditions."

9. Updates to Chapter 6, Volume 2D

Change bars and violet text show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D:* Instruction Set Reference, W-Z.

Changes to this chapter:

- Revised opcode tables removing REX+ prefixes for instructions: XADD, XCHG, XOR.
- Removed footnote references to verify vector options for the following instructions:
  - XORPD
  - XORPS

CHAPTER 6
INSTRUCTION SET REFERENCE, W-Z

6.1 INSTRUCTIONS (W-Z)

Chapter 6 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (W-Z). See also: Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A; Chapter 4, "Instruction Set Reference, M-U," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B; and Chapter 5, "Instruction Set Reference, V," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

XADD—Exchange and Add

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C0 /r	XADD r/m8[1], r8[1]	MR	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
0F C1 /r	XADD r/m16, r16	MR	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
0F C1 /r	XADD r/m32, r32	MR	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + 0F C1 /r	XADD r/m64, r64	MR	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r, w)	N/A	N/A

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

```
TEMP := SRC + DEST;
SRC := DEST;
DEST := TEMP;
```

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XCHG—Exchange Register/Memory With Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90+rw	XCHG AX, r16	O	Valid	Valid	Exchange r16 with AX.
90+rw	XCHG r16, AX	O	Valid	Valid	Exchange AX with r16.
90+rd	XCHG EAX, r32	O	Valid	Valid	Exchange r32 with EAX.
REX.W + 90+rd	XCHG RAX, r64	O	Valid	N.E.	Exchange r64 with RAX.
90+rd	XCHG r32, EAX	O	Valid	Valid	Exchange EAX with r32.
REX.W + 90+rd	XCHG r64, RAX	O	Valid	N.E.	Exchange RAX with r64.
86 /r	XCHG r/m8[1], r8[1]	MR	Valid	Valid	Exchange r8 (byte register) with byte from r/m8.
86 /r	XCHG r8[1], r/m8[1]	RM	Valid	Valid	Exchange byte from r/m8 with r8 (byte register).
87 /r	XCHG r/m16, r16	MR	Valid	Valid	Exchange r16 with word from r/m16.
87 /r	XCHG r16, r/m16	RM	Valid	Valid	Exchange word from r/m16 with r16.
87 /r	XCHG r/m32, r32	MR	Valid	Valid	Exchange r32 with doubleword from r/m32.
REX.W + 87 /r	XCHG r/m64, r64	MR	Valid	N.E.	Exchange r64 with quadword from r/m64.
87 /r	XCHG r32, r/m32	RM	Valid	Valid	Exchange doubleword from r/m32 with r32.
REX.W + 87 /r	XCHG r64, r/m64	RM	Valid	N.E.	Exchange quadword from r/m64 with r64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	AX/EAX/RAX (r, w)	opcode + rd (r, w)	N/A	N/A
O	opcode + rd (r, w)	AX/EAX/RAX (r, w)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

NOTE

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation

```
TEMP := DEST;
DEST := SRC;
SRC := TEMP;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If either operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XOR—Logical Exclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
34 ib	XOR AL, imm8	I	Valid	Valid	AL XOR imm8.
35 iw	XOR AX, imm16	I	Valid	Valid	AX XOR imm16.
35 id	XOR EAX, imm32	I	Valid	Valid	EAX XOR imm32.
REX.W + 35 id	XOR RAX, imm32	I	Valid	N.E.	RAX XOR imm32 (sign-extended).
80 /6 ib	XOR r/m8[1], imm8	MI	Valid	Valid	r/m8 XOR imm8.
81 /6 iw	XOR r/m16, imm16	MI	Valid	Valid	r/m16 XOR imm16.
81 /6 id	XOR r/m32, imm32	MI	Valid	Valid	r/m32 XOR imm32.
REX.W + 81 /6 id	XOR r/m64, imm32	MI	Valid	N.E.	r/m64 XOR imm32 (sign-extended).
83 /6 ib	XOR r/m16, imm8	MI	Valid	Valid	r/m16 XOR imm8 (sign-extended).
83 /6 ib	XOR r/m32, imm8	MI	Valid	Valid	r/m32 XOR imm8 (sign-extended).
REX.W + 83 /6 ib	XOR r/m64, imm8	MI	Valid	N.E.	r/m64 XOR imm8 (sign-extended).
30 /r	XOR r/m8[1], r8[1]	MR	Valid	Valid	r/m8 XOR r8.
31 /r	XOR r/m16, r16	MR	Valid	Valid	r/m16 XOR r16.
31 /r	XOR r/m32, r32	MR	Valid	Valid	r/m32 XOR r32.
REX.W + 31 /r	XOR r/m64, r64	MR	Valid	N.E.	r/m64 XOR r64.
32 /r	XOR r8,[1] r/m8[1]	RM	Valid	Valid	r8 XOR r/m8.
33 /r	XOR r16, r/m16	RM	Valid	Valid	r16 XOR r/m16.
33 /r	XOR r32, r/m32	RM	Valid	Valid	r32 XOR r/m32.
REX.W + 33 /r	XOR r64, r/m64	RM	Valid	N.E.	r64 XOR r/m64.

NOTES:

1. With a REX prefix in 64-bit mode, attempts to access AH, BH, CH, or DH will instead access SPL, DIL, BPL, or SIL, respectively.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST XOR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical XOR of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 57 /r VXORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 57 /r VXORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 57 /r VXORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical XOR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical XOR of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with write-mask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[63:0];
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VXORPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE XOR SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE XOR SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VXORPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

XORPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE XOR SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE XOR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VXORPD __m512d _mm512_xor_pd (__m512d a, __m512d b);
VXORPD __m512d _mm512_mask_xor_pd (__m512d a, __mmask8 m, __m512d b);
VXORPD __m512d _mm512_maskz_xor_pd (__mmask8 m, __m512d a);
VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);
VXORPD __m256d _mm256_mask_xor_pd (__m256d a, __mmask8 m, __m256d b);
VXORPD __m256d _mm256_maskz_xor_pd (__mmask8 m, __m256d a);
XORPD __m128d _mm_xor_pd (__m128d a, __m128d b);
VXORPD __m128d _mm_mask_xor_pd (__m128d a, __mmask8 m, __m128d b);
VXORPD __m128d _mm_maskz_xor_pd (__mmask8 m, __m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 57 /r XORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical XOR of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F.WIG 57 /r VXORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	(AVX512VL AND AVX512DQ) OR AVX10.1	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ OR AVX10.1	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with write-mask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
          IF (EVEX.b == 1) AND (SRC2 *is memory*)
             THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[31:0];
             ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];
          FI;
      ELSE
          IF *merging-masking*                    ; merging-masking
             THEN *DEST[i+31:i] remains unchanged*
             ELSE *zeroing-masking*               ; zeroing-masking
                DEST[i+31:i] = 0
          FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VXORPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE XOR SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE XOR SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE XOR SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE XOR SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VXORPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

XORPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VXORPS __m512 _mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 _mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 _mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 _mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 _mm256_maskz_xor_ps (__mmask8 m, __m256 a);
XORPS __m128 _mm_xor_ps (__m128 a, __m128 b);
VXORPS __m128 _mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
VXORPS __m128 _mm_maskz_xor_ps (__mmask8 m, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

10. Updates to Chapter 17, Volume 3B

Change bars and violet text show changes to Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter:

- In section Section 17.3.2.4, "IA32_MCi_MISC MSRs," removed mention of Intel® Atom processors, which implements the IA32_MCi_STATUS register in error-reporting register banks.

CHAPTER 17
MACHINE-CHECK ARCHITECTURE

This chapter describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. See Chapter 7, "Interrupt 18—Machine-Check Exception (#MC)," for more information on machine-check exceptions. A brief description of the Pentium processor's machine check capability is also given.

Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

17.1 MACHINE-CHECK ARCHITECTURE

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors implement a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as: system bus errors, ECC errors, parity errors, cache errors, and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs used for recording errors that are detected.

The processor signals the detection of an uncorrected machine-check error by generating a machine-check exception (#MC), which is an abort class exception. The implementation of the machine-check architecture does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check-exception handler can collect information about the machine-check error from the machine-check MSRs.

Starting with 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH; see the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. The processor can report information on corrected machine-check errors and deliver a programmable interrupt for software to respond to MC errors, referred to as corrected machine-check error interrupt (CMCI). See Section 17.5 for details.

Intel 64 processors supporting machine-check architecture and CMCI may also support an additional enhancement, namely, support for software recovery from certain uncorrected recoverable machine check errors. See Section 17.6 for details.

17.2 COMPATIBILITY WITH PENTIUM PROCESSOR

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support and extend the machine-check exception mechanism introduced in the Pentium processor. The Pentium processor reports the following machine-check errors:

- Data parity errors during read cycles.
- Unsuccessful completion of a bus cycle.

The above errors are reported using the P5_MC_TYPE and P5_MC_ADDR MSRs (implementation specific for the Pentium processor). Use the RDMSR instruction to read these MSRs. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for the addresses.

The machine-check error reporting mechanism that Pentium processors use is similar to that used in Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. When an error is detected, it is recorded in P5_MC_TYPE and P5_MC_ADDR; the processor then generates a machine-check exception (#MC).

See Section 17.3.3, "Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture," and Section 17.10.2, "Pentium Processor Machine-Check Exception Handling," for information on compatibility between machine-check code written to run on the Pentium processors and code written to run on P6 family processors.

17.3 MACHINE-CHECK MSRS

Machine check MSRs in the Pentium 4, Intel Atom, Intel Xeon, and P6 family processors consist of a set of global control and status registers and several error-reporting register banks. See Figure 17-1.

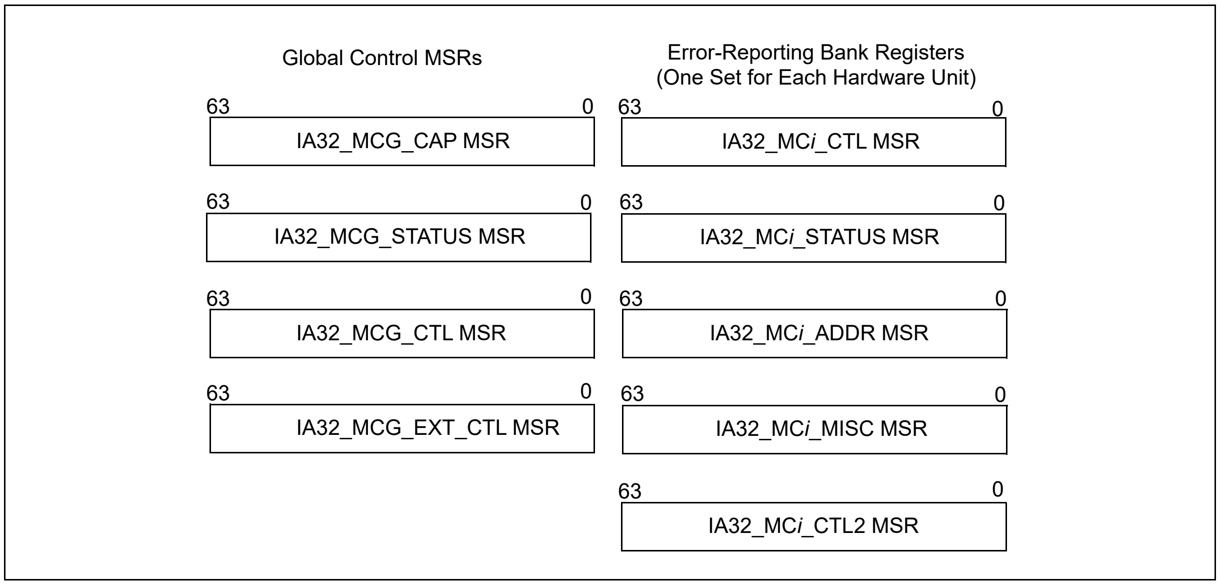


Figure 17-1. Machine-Check MSRs

Each error-reporting bank is associated with a specific hardware unit (or group of hardware units) in the processor. Use RDMSR and WRMSR to read and to write these registers.

17.3.1 Machine-Check Global Control MSRs

The machine-check global control MSRs include the IA32_MCG_CAP, IA32_MCG_STATUS, and optionally IA32_MC-G_CTL and IA32_MCG_EXT_CTL. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for the addresses of these registers.

17.3.1.1 IA32_MCG_CAP MSR

The IA32_MCG_CAP MSR is a read-only register that provides information about the machine-check architecture of the processor. Figure 17-2 shows the layout of the register.

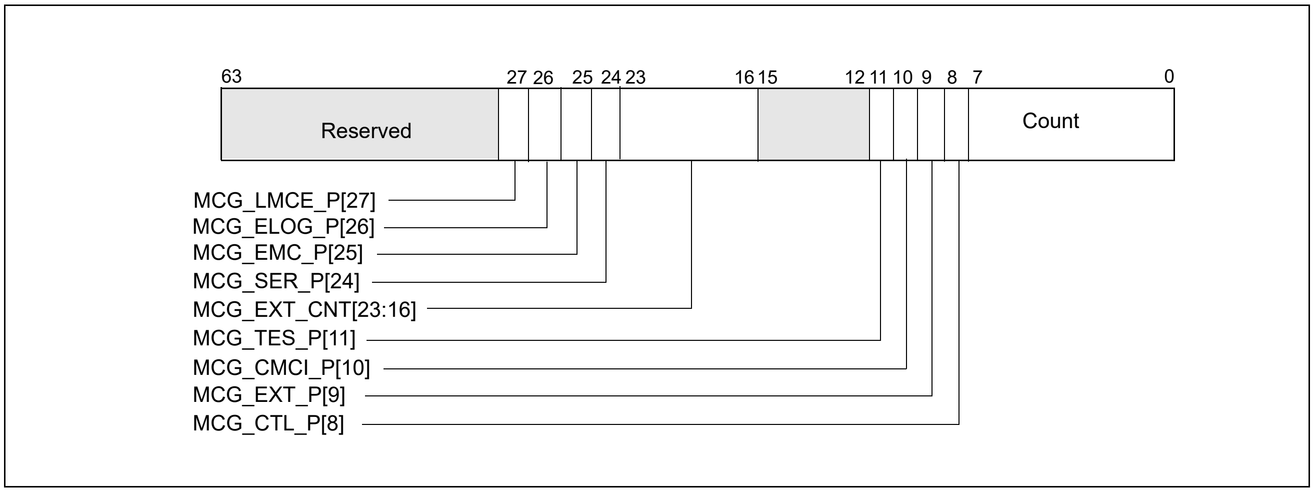


Figure 17-2. IA32_MCG_CAP Register

Where:

- **Count field, bits 7:0** — Indicates the number of hardware unit error-reporting banks available in a particular processor implementation.
- **MCG_CTL_P (control MSR present) flag, bit 8** — Indicates that the processor implements the IA32_MCG_CTL MSR when set; this register is absent when clear.
- **MCG_EXT_P (extended MSRs present) flag, bit 9** — Indicates that the processor implements the extended machine-check state registers found starting at MSR address 180H; these registers are absent when clear.
- **MCG_CMCI_P (Corrected MC error counting/signaling extension present) flag, bit 10** — Indicates (when set) that extended state and associated MSRs necessary to support the reporting of an interrupt on a corrected MC error event and/or count threshold of corrected MC errors, is present. When this bit is set, it does not imply this feature is supported across all banks. Software should check the availability of the necessary logic on a bank by bank basis when using this signaling capability (i.e., bit 30 settable in individual IA32_MCi_CTL2 register).
- **MCG_TES_P (threshold-based error status present) flag, bit 11** — Indicates (when set) that bits 56:53 of the IA32_MCi_STATUS MSR are part of the architectural space. Bits 56:55 are reserved, and bits 54:53 are used to report threshold-based error status. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific.
- **MCG_EXT_CNT, bits 23:16** — Indicates the number of extended machine-check state registers present. This field is meaningful only when the MCG_EXT_P flag is set.
- **MCG_SER_P (software error recovery support present) flag, bit 24** — Indicates (when set) that the processor supports software error recovery (see Section 17.6), and IA32_MCi_STATUS MSR bits 56:55 are used to report the signaling of uncorrected recoverable errors and whether software must take recovery actions for uncorrected errors. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific. If MCG_TES_P is set but MCG_SER_P is not set, bits 56:55 are reserved.
- **MCG_EMC_P (Enhanced Machine Check Capability) flag, bit 25** — Indicates (when set) that the processor supports enhanced machine check capabilities for firmware first signaling.
- **MCG_ELOG_P (extended error logging) flag, bit 26** — Indicates (when set) that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format "Generic Error Data Entry" that augments the data included in machine check bank registers.

  For additional information about extended error logging interface, see https://cdrdv2.intel.com/v1/dl/getContent/671064.
- **MCG_LMCE_P (local machine check exception) flag, bit 27** — Indicates (when set) that the following interfaces are present:

— an extended state LMCE_S (located in bit 3 of IA32_MCG_STATUS), and

— the IA32_MCG_EXT_CTL MSR, necessary to support Local Machine Check Exception (LMCE).

A non-zero MCG_LMCE_P indicates that, when LMCE is enabled as described in Section 17.3.1.5, some machine check errors may be delivered to only a single logical processor.

The effect of writing to the IA32_MCG_CAP MSR is undefined.

17.3.1.2 IA32_MCG_STATUS MSR

The IA32_MCG_STATUS MSR describes the current state of the processor after a machine-check exception has occurred (see Figure 17-3).

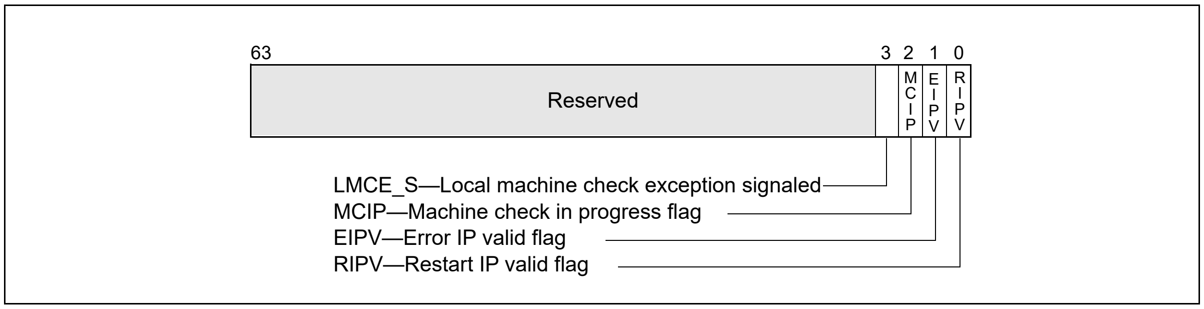

63
3
2
1
0
Reserved
MCIP
EIPV
RIPV
LMCE_S—Local machine check exception signaled
MCIP—Machine check in progress flag
EIPV—Error IP valid flag
RIPV—Restart IP valid flag

Figure 17-3. IA32_MCG_STATUS Register

Where:

- **RIPV (restart IP valid) flag, bit 0** — Indicates (when set) that program execution can be restarted reliably at the instruction pointed to by the instruction pointer pushed on the stack when the machine-check exception is generated. When clear, the program cannot be reliably restarted at the pushed instruction pointer.
- **EIPV (error IP valid) flag, bit 1** — Indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.
- **MCIP (machine check in progress) flag, bit 2** — Indicates (when set) that a machine-check exception was generated. Software can set or clear this flag. The occurrence of a second Machine-Check Event while MCIP is set will cause the processor to enter a shutdown state. For information on processor behavior in the shutdown state, please refer to the description in Chapter 7, "Interrupt and Exception Handling": "Interrupt 8—Double Fault Exception (#DF)".
- **LMCE_S (local machine check exception signaled), bit 3** — Indicates (when set) that a local machine-check exception was generated. This indicates that the current machine-check event was delivered to only this logical processor.

Bits 63:04 in the IA32_MCG_STATUS MSR are reserved. An attempt to write to the IA32_MCG_STATUS MSR's reserved bits with any value other than 0 results in #GP.

17.3.1.3 IA32_MCG_CTL MSR

The IA32_MCG_CTL MSR is present if the capability flag MCG_CTL_P is set in the IA32_MCG_CAP MSR.

IA32_MCG_CTL controls the reporting of machine-check exceptions. If present, writing 1s to this register enables machine-check features and writing all 0s disables machine-check features. All other values are undefined and/or implementation specific.

17.3.1.4 IA32_MCG_EXT_CTL MSR

The IA32_MCG_EXT_CTL MSR is present if the capability flag MCG_LMCE_P is set in the IA32_MCG_CAP MSR.

IA32_MCG_EXT_CTL.LMCE_EN (bit 0) allows the processor to signal some MCEs to only a single logical processor in the system.

If MCG_LMCE_P is not set in IA32_MCG_CAP, or platform software has not enabled LMCE by setting IA32_FEATURE_CONTROL.LMCE_ENABLED (bit 20), any attempt to write or read IA32_MCG_EXT_CTL will result in #GP.

The IA32_MCG_EXT_CTL MSR is cleared on RESET.

Figure 17-4 shows the layout of the IA32_MCG_EXT_CTL register

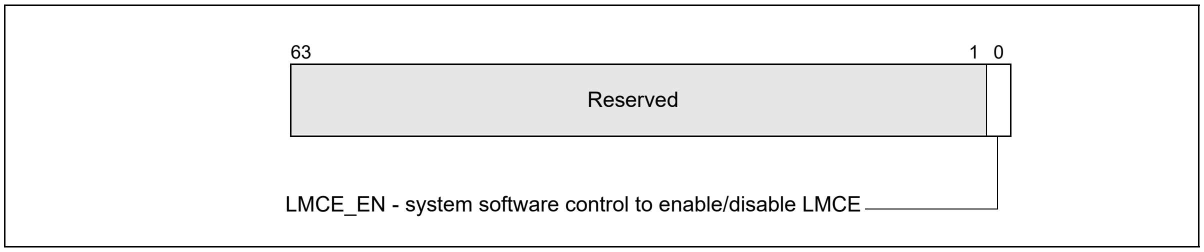


Figure 17-4. IA32_MCG_EXT_CTL Register

where

- **LMCE_EN (local machine check exception enable) flag, bit 0** - System software sets this to allow hardware to signal some MCEs to only a single logical processor. System software can set LMCE_EN only if the platform software has configured IA32_FEATURE_CONTROL as described in Section 17.3.1.5.

17.3.1.5 Enabling Local Machine Check

The intended usage of LMCE requires proper configuration by both platform software and system software. Platform software can turn LMCE on by setting bit 20 (LMCE_ENABLED) in IA32_FEATURE_CONTROL MSR (MSR address 3AH).

System software must ensure that both IA32_FEATURE_CONTROL.Lock (bit 0)and IA32_FEATURE_CONTROL.LMCE_ENABLED (bit 20) are set before attempting to set IA32_MCG_EXT_CTL.LMCE_EN (bit 0). When system software has enabled LMCE, then hardware will determine if a particular error can be delivered only to a single logical processor. Software should make no assumptions about the type of error that hardware can choose to deliver as LMCE. The severity and override rules stay the same as described in Table 17-8 to determine the recovery actions.

17.3.2 Error-Reporting Register Banks

Each error-reporting register bank can contain the IA32_MCi_CTL, IA32_MCi_STATUS, IA32_MCi_ADDR, and IA32_MCi_MISC MSRs. The number of reporting banks is indicated by bits [7:0] of IA32_MCG_CAP MSR (address 0179H). The first error-reporting register (IA32_MC0_CTL) always starts at address 400H.

See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for addresses of the error-reporting registers in the Pentium 4, Intel Atom, and Intel Xeon processors; and for addresses of the error-reporting registers P6 family processors.

17.3.2.1 IA32_MCi_CTL MSRs

The IA32_MC*i*_CTL MSR controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units). Each of the 64 flags (EE*j*) represents a potential error. Setting an EE*j* flag enables signaling #MC of the associated error and clearing it disables signaling of the error. Error logging happens regardless of the setting of these bits. The processor drops writes to bits that are not implemented. Figure 17-5 shows the bit fields of IA32_MC*i*_CTL.

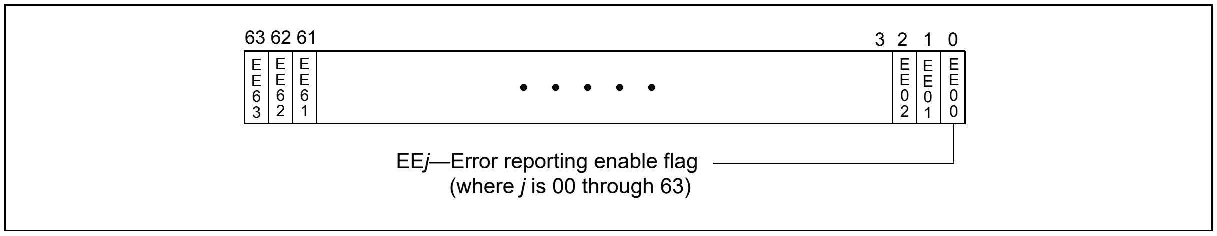


Figure 17-5. IA32_MC*i*_CTL Register

NOTE

For P6 family processors, processors based on Intel Core microarchitecture (excluding those on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH and onward): the operating system or executive software must not modify the contents of the IA32_MC0_CTL MSR. This MSR is internally aliased to the EBL_CR_POWERON MSR and controls platform-specific error handling features. System specific firmware (the BIOS) is responsible for the appropriate initialization of the IA32_MC0_CTL MSR. P6 family processors only allow the writing of all 1s or all 0s to the IA32_MC*i*_CTL MSR.

17.3.2.2 IA32_MCi_STATUS MSRS

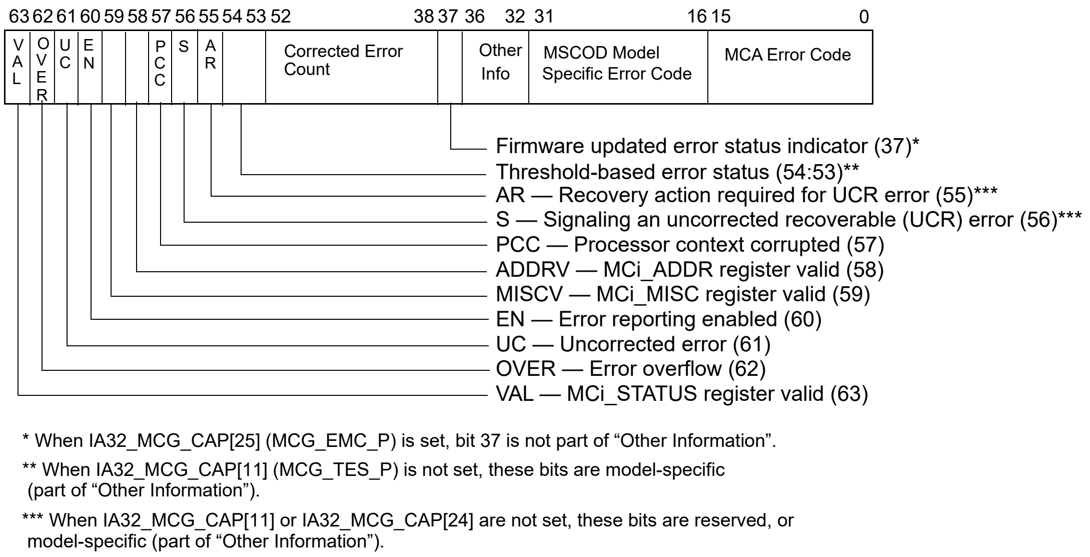
Each IA32_MCi_STATUS MSR contains information related to a machine-check error if its VAL (valid) flag is set (see Figure 17-6). Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.

NOTE

Figure 17-6 depicts the IA32_MCi_STATUS MSR when IA32_MCG_CAP[24] = 1, IA32_MCG_CAP[11] = 1 and IA32_MCG_CAP[10] = 1. When IA32_MCG_CAP[24] = 0 and IA32_MCG_CAP[11] = 1, bits 56:55 is reserved and bits 54:53 for threshold-based error reporting. When IA32_MCG_CAP[11] = 0, bits 56:53 are part of the "Other Information" field. The use of bits 54:53 for threshold-based error reporting began with Intel Core Duo processors, and is currently used for cache memory. See Section 17.4, "Enhanced Cache Error reporting," for more information. When IA32_MCG_CAP[10] = 0, bits 52:38 are part of the "Other Information" field. The use of bits 52:38 for corrected MC error count is introduced with Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH.

Where:

- **MCA (machine-check architecture) error code field, bits 15:0** — Specifies the machine-check architecture-defined error code for the machine-check error condition detected. The machine-check architecture-defined error codes are guaranteed to be the same for all IA-32 processors that implement the machine-check architecture. See Section 17.9, "Interpreting the MCA Error Codes," and Chapter 18, "Interpreting Machine Check Error Codes," for information on machine-check error codes.
- **Model-specific error code field, bits 31:16** — Specifies the model-specific error code that uniquely identifies the machine-check error condition detected. The model-specific error codes may differ among IA-32 processors for the same machine-check error condition. See Chapter 18, "Interpreting Machine Check Error Codes," for information on model-specific error codes.
- **Reserved, Error Status, and Other Information fields, bits 56:32** —
    - If IA32_MCG_CAP.MCG_EMC_P[bit 25] is 0, bits 37:32 contain "Other Information" that is implementation-specific and is not part of the machine-check architecture.
    - If IA32_MCG_CAP.MCG_EMC_P is 1, "Other Information" is in bits 36:32. If bit 37 is 0, system firmware has not changed the contents of IA32_MCi_STATUS. If bit 37 is 1, system firmware may have edited the contents of IA32_MCi_STATUS.
    - If IA32_MCG_CAP.MCG_CMCI_P[bit 10] is 0, bits 52:38 also contain "Other Information" (in the same sense as bits 37:32).



63 62 61 60 59 58 57 56 55 54 53 52
38 37 36 32 31 16 15 0
VAL
OVER
UC
EN
PCC
S
AR
Corrected Error Count
Other Info
MSCOD Model Specific Error Code
MCA Error Code
Firmware updated error status indicator (37)*
Threshold-based error status (54:53)**
AR — Recovery action required for UCR error (55)***
S — Signaling an uncorrected recoverable (UCR) error (56)***
PCC — Processor context corrupted (57)
ADDRV — MCi_ADDR register valid (58)
MISCV — MCi_MISC register valid (59)
EN — Error reporting enabled (60)
UC — Uncorrected error (61)
OVER — Error overflow (62)
VAL — MCi_STATUS register valid (63)
* When IA32_MCG_CAP[25] (MCG_EMC_P) is set, bit 37 is not part of "Other Information".
** When IA32_MCG_CAP[11] (MCG_TES_P) is not set, these bits are model-specific (part of "Other Information").
*** When IA32_MCG_CAP[11] or IA32_MCG_CAP[24] are not set, these bits are reserved, or model-specific (part of "Other Information").

Figure 17-6. IA32_MC*i*_STATUS Register

- If IA32_MCG_CAP[10] is 1, bits 52:38 are architectural (not model-specific). In this case, bits 52:38 reports the value of a 15 bit counter that increments each time a corrected error is observed by the MCA recording bank. This count value will continue to increment until cleared by software. The most significant bit, 52, is a sticky count overflow bit.
- If IA32_MCG_CAP[11] is 0, bits 56:53 also contain "Other Information" (in the same sense).
- If IA32_MCG_CAP[11] is 1, bits 56:53 are architectural (not model-specific). In this case, bits 56:53 have the following functionality:
  - If IA32_MCG_CAP[24] is 0, bits 56:55 are reserved.
  - If IA32_MCG_CAP[24] is 1, bits 56:55 are defined as follows:
  - S (Signaling) flag, bit 56 - Signals the reporting of UCR errors in this MC bank. See Section 17.6.2 for additional details.
  - AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. See Section 17.6.2 for additional details.
  - If the UC bit (Figure 17-6) is 1, bits 54:53 are undefined.
  - If the UC bit (Figure 17-6) is 0, bits 54:53 indicate the status of the hardware structure that reported the threshold-based error. See Table 17-1.

Table 17-1. Bits 54:53 in IA32_MCi_STATUS MSRs when IA32_MCG_CAP[11] = 1 and UC = 0

Bits 54:53	Meaning
00	**No tracking** - No hardware status tracking is provided for the structure reporting this event.
01	**Green** - Status tracking is provided for the structure posting the event; the current status is green (below threshold). For more information, see Section 17.4, "Enhanced Cache Error reporting."
10	**Yellow** - Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). For more information, see Section 17.4, "Enhanced Cache Error reporting."
11	Reserved

- **PCC (processor context corrupt) flag, bit 57** — Indicates (when set) that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor's state, and software may be able to restart. When system software supports recovery, consult Section 17.10.4, "Machine-Check Software Handler Guidelines for Error Recovery," for additional rules that apply.
- **ADDRV (IA32_MC*i*_ADDR register valid) flag, bit 58** — Indicates (when set) that the IA32_MCi_ADDR register contains the address where the error occurred (see Section 17.3.2.3, "IA32_MCi_ADDR MSRs"). When clear, this flag indicates that the IA32_MCi_ADDR register is either not implemented or does not contain the address where the error occurred. Do not read these registers if they are not implemented in the processor.
- **MISCV (IA32_MC*i*_MISC register valid) flag, bit 59** — Indicates (when set) that the IA32_MC*i*_MISC register contains additional information regarding the error. When clear, this flag indicates that the IA32_M-C*i*_MISC register is either not implemented or does not contain additional information regarding the error. Do not read these registers if they are not implemented in the processor.
- **EN (error enabled) flag, bit 60** — Indicates (when set) that the error was enabled by the associated EEj bit of the IA32_MC*i*_CTL register.
- **UC (error uncorrected) flag, bit 61** — Indicates (when set) that the processor did not or was not able to correct the error condition. When clear, this flag indicates that the processor was able to correct the error condition.
- **OVER (machine check overflow) flag, bit 62** — Indicates (when set) that a machine-check error occurred while the results of a previous error were still in the error-reporting register bank (that is, the VAL bit was already set in the IA32_MC*i*_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. In general, enabled errors are written over disabled errors, and uncorrected errors are written over corrected errors. Uncorrected errors are not written over previous valid uncorrected errors. When MCG_CMCI_P is set, corrected errors may not set the OVER flag. Software can rely on corrected error count in IA32_MCi_Status[52:38] to determine if any additional corrected errors may have occurred. For more information, see Section 17.3.2.2.1, "Overwrite Rules for Machine Check Overflow."
- **VAL (IA32_MC*i*_STATUS register valid) flag, bit 63** — Indicates (when set) that the information within the IA32_MCi_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the IA32_MCi_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

17.3.2.2.1 Overwrite Rules for Machine Check Overflow

Table 17-2 shows the overwrite rules for how to treat a second event if the MC bank already contains a valid log from an earlier event – that is, what to do if the valid bit for an MC bank already is set to 1. When more than one structure posts events in a given bank, these rules specify whether a new event will overwrite a previous posting or not. These rules define a priority for uncorrected (highest priority), yellow, and green/unmonitored (lowest priority) status.

In Table 17-2, the values in the two left-most columns are IA32_MCi_STATUS[54:53].

Table 17-2. Overwrite Rules for Enabled Errors

First Event	Second Event	UC bit	Color	MCA Info
00/green	00/green	0	00/green	either
00/green	yellow	0	yellow	second error
yellow	00/green	0	yellow	first error
yellow	yellow	0	yellow	either
00/green/yellow	UC	1	undefined	second
UC	00/green/yellow	1	undefined	first

If a second event overwrites a previously posted event, the information (as guarded by individual valid bits) in the MCi bank is entirely from the second event. Similarly, if a first event is retained, all of the information previously posted for that event is retained. In general, when the logged error or the recent error is a corrected error, the OVER bit (MCi_Status[62]) may be set to indicate an overflow. When MCG_CMCI_P is set in IA32_MCG_CAP, system software should consult IA32_MCi_STATUS[52:38] to determine if additional corrected errors may have occurred. Software may re-read IA32_MCi_STATUS, IA32_MCi_ADDR, and IA32_MCi_MISC appropriately to ensure data collected represent the last error logged.

After software polls a posting and clears the register, the valid bit is no longer set and therefore the meaning of the rest of the bits, including the yellow/green/00 status field in bits 54:53, is undefined. The yellow/green indication will only be posted for events associated with monitored structures – otherwise the unmonitored (00) code will be posted in IA32_MC*i*_STATUS[54:53].

17.3.2.3 IA32_MCi_ADDR MSRs

The IA32_MC*i*_ADDR MSR contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MC*i*_STATUS register is set (see Section 17-7, "IA32_MCi_ADDR MSR"). The IA32_MCi_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_M-Ci_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception.

The address returned is an offset into a segment, linear address, or physical address. This depends on the error encountered. When these registers are implemented, these registers can be cleared by explicitly writing 0s to these registers. Writing 1s to these registers will cause a general-protection exception. See Figure 17-7.

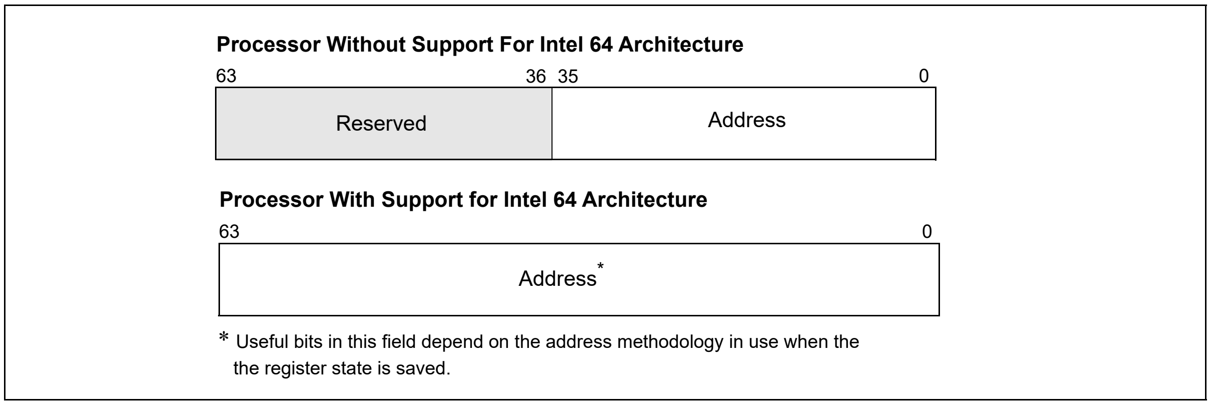

Processor Without Support For Intel 64 Architecture
63
36
35
0
Reserved
Address
Processor With Support for Intel 64 Architecture
63
0
Address*
* Useful bits in this field depend on the address methodology in use when the the register state is saved.

Figure 17-7. IA32_MC*i*_ADDR MSR

17.3.2.4 IA32_MCi_MISC MSRs

The IA32_MC*i*_MISC MSR contains additional information describing the machine-check error if the MISCV flag in the IA32_MC*i*_STATUS register is set. The IA32_MCi_MISC_MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MCi_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception. When implemented in a processor, these registers can be cleared by explicitly writing all bits to 0; writing a 1 to any bit causes a general-protection exception to be generated. This register is not implemented in any of the error-reporting register banks for the Intel P6 family processors.

If both MISCV and IA32_MCG_CAP[24] are set, the IA32_MCi_MISC_MSR is defined according to Figure 17-8 to support software recovery of uncorrected errors (see Section 17.6).

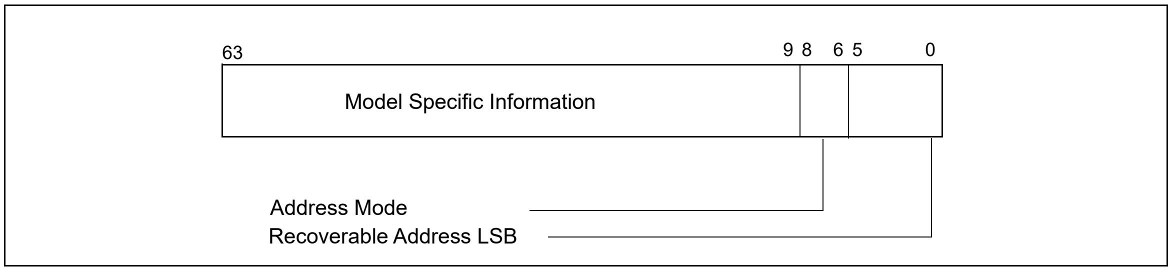


Figure 17-8. UCR Support in IA32_MCi_MISC Register

- Recoverable Address LSB (bits 5:0): The lowest valid recoverable address bit. Indicates the position of the least significant bit (LSB) of the recoverable error address. For example, if the processor logs bits [43:9] of the address, the LSB sub-field in IA32_MCi_MISC is 01001b (9 decimal). For this example, bits [8:0] of the recoverable error address in IA32_MCi_ADDR should be ignored.
- Address Mode (bits 8:6): Address mode for the address logged in IA32_MCi_ADDR. The supported address modes are given in Table 17-3.

Table 17-3. Address Mode in IA32_MCi_MISC[8:6]

IA32_MCi_MISC[8:6] Encoding	Definition
000	Segment Offset
001	Linear Address
010	Physical Address
011	Memory Address
100 to 110	Reserved
111	Generic

- Model Specific Information (bits 63:9): Not architecturally defined.

17.3.2.4.1 IOMCA

Logging and Signaling of errors from PCI Express domain is governed by PCI Express Advanced Error Reporting (AER) architecture. PCI Express architecture divides errors in two categories: Uncorrectable errors and Correctable errors. Uncorrectable errors can further be classified as Fatal or Non-Fatal. Uncorrected IO errors are signaled to the system software either as AER Message Signaled Interrupt (MSI) or via platform specific mechanisms such as NMI. Generally, the signaling mechanism is controlled by BIOS and/or platform firmware. Certain processors support an error handling mode, called IOMCA mode, where Uncorrected PCI Express errors are signaled in the form of machine check exception and logged in machine check banks.

When a processor is in this mode, Uncorrected PCI Express errors are logged in the MCACOD field of the IA32_MCi_STATUS register as Generic I/O error. The corresponding MCA error code is defined in Table 15-8. IA32_MCi_Status [15:0] Simple Error Code Encoding. Machine check logging complements and does not replace AER logging that occurs inside the PCI Express hierarchy. The PCI Express Root Complex and Endpoints continue to log the error in accordance with PCI Express AER mechanism. In IOMCA mode, MCi_MISC register in the bank that logged IOMCA can optionally contain information that link the Machine Check logs with the AER logs or proprietary logs. In such a scenario, the machine check handler can utilize the contents of MCi_MISC to locate the next level of error logs corresponding to the same error. Specifically, if MCi_Status.MISCV is 1 and MCACOD is 0x0E0B, MCi_MISC contains the PCI Express address of the Root Complex device containing the AER Logs. Software can consult the header type and class code registers in the Root Complex device's PCIe Configuration space to determine what type of device it is. This Root Complex device can either be a PCI Express Root Port, PCI Express Root Complex Event Collector or a proprietary device.

Errors that originate from PCI Express or Legacy Endpoints are logged in the corresponding Root Port in addition to the generating device. If MISCV=1 and MCi_MISC contains the address of the Root Port or a Root Complex Event collector, software can parse the AER logs to learn more about the error.

If MISCV=1 and MCi_MISC points to a device that is neither a Root Complex Event Collector not a Root Port, software must consult the Vendor ID/Device ID and use device specific knowledge to locate and interpret the error log registers. In some cases, the Root Complex device configuration space may not be accessible to the software and both the Vendor and Device ID read as 0xFFFF.

- The format of MCi_MISC for IOMCA errors is shown in Table 17-4.

Table 17-4. Address Mode in IA32_MCi_MISC[8:6]

63:40	39:32	31:16	15:9	8:6	5:0
RSVD	PCI Express Segment number	PCI Express Requestor ID	RSVD	ADDR MODE[1]	RECOV ADDR LSB[1]

NOTES:

1. Not Applicable if ADDRV=0.

Refer to PCI Express Specification 3.0 for definition of PCI Express Requestor ID and AER architecture. Refer to PCI Firmware Specification 3.0 for an explanation of PCI Ex-press Segment number and how software can access configuration space of a PCI Ex-press device given the segment number and Requestor ID.

17.3.2.5 IA32_MCi_CTL2 MSRs

The IA32_MC*i*_CTL2 MSR provides the programming interface to use corrected MC error signaling capability that is indicated by IA32_MCG_CAP[10] = 1. Software must check for the presence of IA32_MC*i*_CTL2 on a per-bank basis.

When IA32_MCG_CAP[10] = 1, the IA32_MCi_CTL2 MSR for each bank exists, i.e., reads and writes to these MSR are supported. However, signaling interface for corrected MC errors may not be supported in all banks.

The layout of IA32_MC*i*_CTL2 is shown in Figure 17-9.

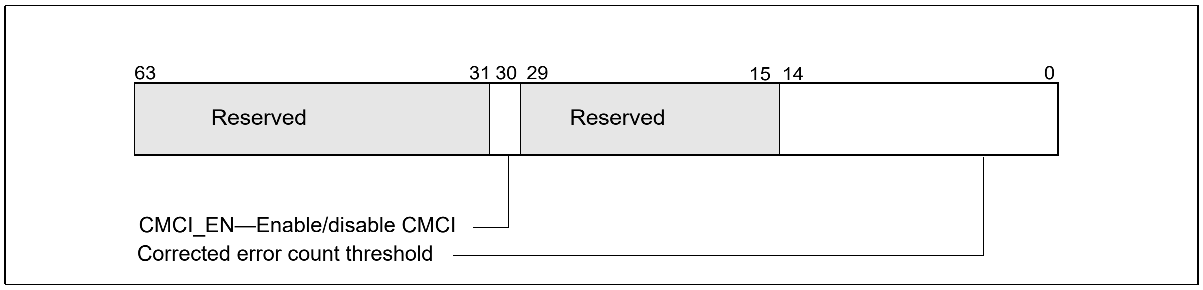


Figure 17-9. IA32_MCi_CTL2 Register

- **Corrected error count threshold, bits 14:0** — Software must initialize this field. The value is compared with the corrected error count field in IA32_MCi_STATUS, bits 38 through 52. An overflow event is signaled to the CMCI LVT entry (see Table 12-1) in the APIC when the count value equals the threshold value. The new LVT entry in the APIC is at 02F0H offset from the APIC_BASE. If CMCI interface is not supported for a particular bank (but IA32_MCG_CAP[10] = 1), this field will always read 0.
- **CMCI_EN (Corrected error interrupt enable/disable/indicator), bits 30** — Software sets this bit to enable the generation of corrected machine-check error interrupt (CMCI). If CMCI interface is not supported for a particular bank (but IA32_MCG_CAP[10] = 1), this bit is writeable but will always return 0 for that bank. This bit also indicates CMCI is supported or not supported in the corresponding bank. See Section 17.5 for details of software detection of CMCI facility.

Some microarchitectural sub-systems that are the source of corrected MC errors may be shared by more than one logical processors. Consequently, the facilities for reporting MC errors and controlling mechanisms may be shared by more than one logical processors. For example, the IA32_MC*i*_CTL2 MSR is shared between logical processors sharing a processor core. Software is responsible to program IA32_MC*i*_CTL2 MSR in a consistent manner with CMCI delivery and usage.

After processor reset, IA32_MC*i*_CTL2 MSRs are zeroed.

17.3.2.6 IA32_MCG Extended Machine Check State MSRs

The Pentium 4 and Intel Xeon processors implement a variable number of extended machine-check state MSRs. The MCG_EXT_P flag in the IA32_MCG_CAP MSR indicates the presence of these extended registers, and the MCG_EXT_CNT field indicates the number of these registers actually implemented. See Section 17.3.1.1, "IA32_MCG_CAP MSR." Also see Table 17-5.

Table 17-5. Extended Machine Check State MSRs in Processors Without Support for Intel® 64 Architecture

MSR	Address	Description
IA32_MCG_EAX	180H	Contains state of the EAX register at the time of the machine-check error.
IA32_MCG_EBX	181H	Contains state of the EBX register at the time of the machine-check error.
IA32_MCG_ECX	182H	Contains state of the ECX register at the time of the machine-check error.
IA32_MCG_EDX	183H	Contains state of the EDX register at the time of the machine-check error.
IA32_MCG_ESI	184H	Contains state of the ESI register at the time of the machine-check error.
IA32_MCG_EDI	185H	Contains state of the EDI register at the time of the machine-check error.
IA32_MCG_EBP	186H	Contains state of the EBP register at the time of the machine-check error.
IA32_MCG_ESP	187H	Contains state of the ESP register at the time of the machine-check error.
IA32_MCG_EFLAGS	188H	Contains state of the EFLAGS register at the time of the machine-check error.
IA32_MCG_EIP	189H	Contains state of the EIP register at the time of the machine-check error.
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.

In processors with support for Intel 64 architecture, 64-bit machine check state MSRs are aliased to the legacy MSRs. In addition, there may be registers beyond IA32_MCG_MISC. These may include up to five reserved MSRs (IA32_MCG_RESERVED[1:5]) and save-state MSRs for registers introduced in 64-bit mode. See Table 17-6.

Table 17-6. Extended Machine Check State MSRs In Processors With Support for Intel® 64 Architecture

MSR	Address	Description
IA32_MCG_RAX	180H	Contains state of the RAX register at the time of the machine-check error.
IA32_MCG_RBX	181H	Contains state of the RBX register at the time of the machine-check error.
IA32_MCG_RCX	182H	Contains state of the RCX register at the time of the machine-check error.
IA32_MCG_RDX	183H	Contains state of the RDX register at the time of the machine-check error.
IA32_MCG_RSI	184H	Contains state of the RSI register at the time of the machine-check error.
IA32_MCG_RDI	185H	Contains state of the RDI register at the time of the machine-check error.
IA32_MCG_RBP	186H	Contains state of the RBP register at the time of the machine-check error.
IA32_MCG_RSP	187H	Contains state of the RSP register at the time of the machine-check error.
IA32_MCG_RFLAGS	188H	Contains state of the RFLAGS register at the time of the machine-check error.
IA32_MCG_RIP	189H	Contains state of the RIP register at the time of the machine-check error.
IA32_MCG_MISC	18AH	When set, indicates that a page assist or page fault occurred during DS normal operation.

Table 17-6. Extended Machine Check State MSRs In Processors With Support for Intel® 64 Architecture (Contd.)

MSR	Address	Description
IA32_MCG_ RSERVED[1:5]	18BH- 18FH	These registers, if present, are reserved.
IA32_MCG_R8	190H	Contains state of the R8 register at the time of the machine-check error.
IA32_MCG_R9	191H	Contains state of the R9 register at the time of the machine-check error.
IA32_MCG_R10	192H	Contains state of the R10 register at the time of the machine-check error.
IA32_MCG_R11	193H	Contains state of the R11 register at the time of the machine-check error.
IA32_MCG_R12	194H	Contains state of the R12 register at the time of the machine-check error.
IA32_MCG_R13	195H	Contains state of the R13 register at the time of the machine-check error.
IA32_MCG_R14	196H	Contains state of the R14 register at the time of the machine-check error.
IA32_MCG_R15	197H	Contains state of the R15 register at the time of the machine-check error.

When a machine-check error is detected on a Pentium 4 or Intel Xeon processor, the processor saves the state of the general-purpose registers, the R/EFLAGS register, and the R/EIP in these extended machine-check state MSRs. This information can be used by a debugger to analyze the error.

These registers are read/write to zero registers. This means software can read them; but if software writes to them, only all zeros is allowed. If software attempts to write a non-zero value into one of these registers, a general-protection (#GP) exception is generated. These registers are cleared on a hardware reset (power-up or RESET), but maintain their contents following a soft reset (INIT reset).

17.3.3 Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture

The Pentium processor reports machine-check errors using two registers: P5_MC_TYPE and P5_MC_ADDR. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors map these registers to the IA32_MC*i*_STATUS and IA32_MC*i*_ADDR in the error-reporting register bank. This bank reports on the same type of external bus errors reported in P5_MC_TYPE and P5_MC_ADDR.

The information in these registers can then be accessed in two ways:

- By reading the IA32_MC*i*_STATUS and IA32_MC*i*_ADDR registers as part of a general machine-check exception handler written for Pentium 4, Intel Atom and P6 family processors.
- By reading the P5_MC_TYPE and P5_MC_ADDR registers using the RDMSR instruction.

The second capability permits a machine-check exception handler written to run on a Pentium processor to be run on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor. There is a limitation in that information returned by the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors is encoded differently than information returned by the Pentium processor. To run a Pentium processor machine-check exception handler on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor; the handler must be written to interpret P5_MC_TYPE encodings correctly.

17.4 ENHANCED CACHE ERROR REPORTING

Starting with Intel Core Duo processors, cache error reporting was enhanced. In earlier Intel processors, cache status was based on the number of correction events that occurred in a cache. In the new paradigm, called “threshold-based error status”, cache status is based on the number of lines (ECC blocks) in a cache that incur repeated corrections. The threshold is chosen by Intel, based on various factors. If a processor supports threshold-based error status, it sets IA32_MCG_CAP[11] (MCG_TES_P) to 1; if not, to 0.

A processor that supports enhanced cache error reporting contains hardware that tracks the operating status of certain caches and provides an indicator of their “health”. The hardware reports a “green” status when the number of lines that incur repeated corrections is at or below a pre-defined threshold, and a “yellow” status when the number of affected lines exceeds the threshold. Yellow status means that the cache reporting the event is operating correctly, but you should schedule the system for servicing within a few weeks.

Intel recommends that you rely on this mechanism for structures supported by threshold-base error reporting.

The CPU/system/platform response to a yellow event should be less severe than its response to an uncorrected error. An uncorrected error means that a serious error has actually occurred, whereas the yellow condition is a warning that the number of affected lines has exceeded the threshold but is not, in itself, a serious event: the error was corrected and system state was not compromised.

The green/yellow status indicator is not a foolproof early warning for an uncorrected error resulting from the failure of two bits in the same ECC block. Such a failure can occur and cause an uncorrected error before the yellow threshold is reached. However, the chance of an uncorrected error increases as the number of affected lines increases.

17.5 CORRECTED MACHINE CHECK ERROR INTERRUPT

Corrected machine-check error interrupt (CMCI) is an architectural enhancement to the machine-check architecture. It provides capabilities beyond those of threshold-based error reporting (Section 17.4). With threshold-based error reporting, software is limited to use periodic polling to query the status of hardware corrected MC errors. CMCI provides a signaling mechanism to deliver a local interrupt based on threshold values that software can program using the IA32_MCi_CTL2 MSRs.

CMCI is disabled by default. System software is required to enable CMCI for each IA32_MCi bank that support the reporting of hardware corrected errors if IA32_MCG_CAP[10] = 1.

System software use IA32_MCi_CTL2 MSR to enable/disable the CMCI capability for each bank and program threshold values into IA32_MCi_CTL2 MSR. CMCI is not affected by the CR4.MCE bit, and it is not affected by the IA32_MCi_CTL MSRs.

To detect the existence of thresholding for a given bank, software writes only bits 14:0 with the threshold value. If the bits persist, then thresholding is available (and CMCI is available). If the bits are all 0's, then no thresholding exists. To detect that CMCI signaling exists, software writes a 1 to bit 30 of the MCi_CTL2 register. Upon subsequent read, if bit 30 = 0, no CMCI is available for this bank and no corrected or UCNA errors will be reported on this bank. If bit 30 = 1, then CMCI is available and enabled.

17.5.1 CMCI Local APIC Interface

The operation of CMCI is depicted in Figure 17-10.

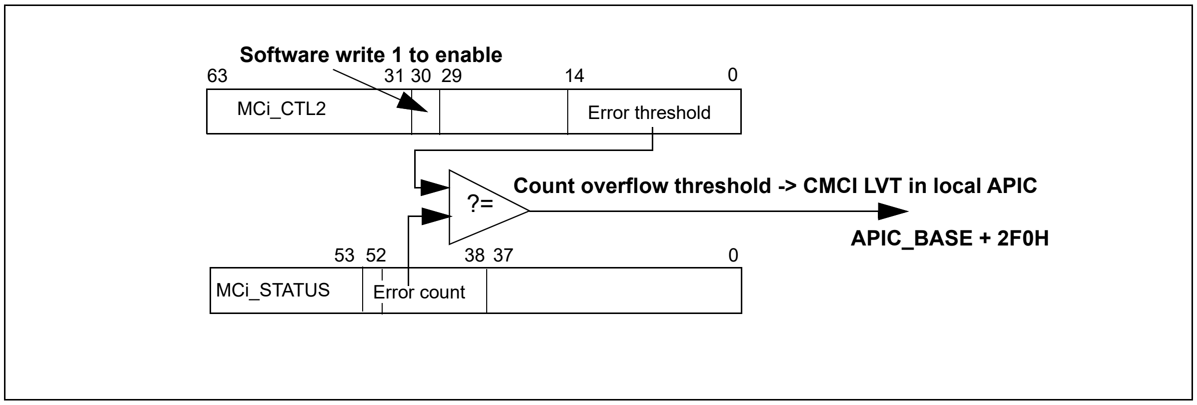


Figure 17-10. CMCI Behavior

CMCI interrupt delivery is configured by writing to the LVT CMCI register entry in the local APIC register space at default address of APIC_BASE + 2F0H. A CMCI interrupt can be delivered to more than one logical processors if multiple logical processors are affected by the associated MC errors. For example, if a corrected bit error in a cache shared by two logical processors caused a CMCI, the interrupt will be delivered to both logical processors sharing that microarchitectural sub-system. Similarly, package level errors may cause CMCI to be delivered to all logical processors within the package. However, system level errors will not be handled by CMCI.

See Section 12.5.1, “Local Vector Table,” for details regarding the LVT CMCI register.

17.5.2 System Software Recommendation for Managing CMCI and Machine Check Resources

System software must enable and manage CMCI, set up interrupt handlers to service CMCI interrupts delivered to affected logical processors, program CMCI LVT entry, and query machine check banks that are shared by more than one logical processors.

This section describes techniques system software can implement to manage CMCI initialization, service CMCI interrupts in a efficient manner to minimize contentions to access shared MSR resources.

17.5.2.1 CMCI Initialization

Although a CMCI interrupt may be delivered to more than one logical processors depending on the nature of the corrected MC error, only one instance of the interrupt service routine needs to perform the necessary service and make queries to the machine-check banks. The following steps describes a technique that limits the amount of work the system has to do in response to a CMCI.

- To provide maximum flexibility, system software should define per-thread data structure for each logical processor to allow equal-opportunity and efficient response to interrupt delivery. Specifically, the per-thread data structure should include a set of per-bank fields to track which machine check bank it needs to access in response to a delivered CMCI interrupt. The number of banks that needs to be tracked is determined by IA32_MCG_CAP[7:0].
- Initialization of per-thread data structure. The initialization of per-thread data structure must be done serially on each logical processor in the system. The sequencing order to start the per-thread initialization between different logical processor is arbitrary. But it must observe the following specific detail to satisfy the shared nature of specific MSR resources:
  a. Each thread initializes its data structure to indicate that it does not own any MC bank registers.
  b. Each thread examines IA32_MCi_CTL2[30] indicator for each bank to determine if another thread has already claimed ownership of that bank.
     - If IA32_MCi_CTL2[30] had been set by another thread. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
     - If IA32_MCi_CTL2[30] = 0, proceed to step c.
  c. Check whether writing a 1 into IA32_MCi_CTL2[30] can return with 1 on a subsequent read to determine this bank can support CMCI.
     - If IA32_MCi_CTL2[30] = 0, this bank does not support CMCI. This thread can not own bank *i* and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
     - If IA32_MCi_CTL2[30] = 1, modify the per-thread data structure to indicate this thread claims ownership to the MC bank; proceed to initialize the error threshold count (bits 15:0) of that bank as described in Chapter 17, “CMCI Threshold Management”. Then proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
- After the thread has examined all of the machine check banks, it sees if it owns any MC banks to service CMCI. If any bank has been claimed by this thread:
  — Ensure that the CMCI interrupt handler has been set up as described in Chapter 17, “CMCI Interrupt Handler”.
  — Initialize the CMCI LVT entry, as described in Section 17.5.1, “CMCI Local APIC Interface.”
  — Log and clear all of IA32_MCi_Status registers for the banks that this thread owns. This will allow new errors to be logged.

17.5.2.2 CMCI Threshold Management

The Corrected MC error threshold field, IA32_MCi_CTL2[14:0], is architecturally defined. Specifically, all these bits are writable by software, but different processor implementations may choose to implement less than 15 bits as threshold for the overflow comparison with IA32_MCi_STATUS[52:38]. The following describes techniques that software can manage CMCI threshold to be compatible with changes in implementation characteristics:

- Software can set the initial threshold value to 1 by writing 1 to IA32_MCi_CTL2[14:0]. This will cause overflow condition on every corrected MC error and generates a CMCI interrupt.
- To increase the threshold and reduce the frequency of CMCI servicing:
  a. Find the maximum threshold value a given processor implementation supports. The steps are:
     - Write 7FFFH to IA32_MCi_CTL2[14:0],
     - Read back IA32_MCi_CTL2[14:0]; these 15 bits (14:0) contain the maximum threshold supported by the processor.

  b. Increase the threshold to a value below the maximum value discovered using step a.

17.5.2.3 CMCI Interrupt Handler

The following describes techniques system software may consider to implement a CMCI service routine:

- The service routine examines its private per-thread data structure to check which set of MC banks it has ownership. If the thread does not have ownership of a given MC bank, proceed to the next MC bank. Ownership is determined at initialization time which is described in Section 17.5.2.1.

If the thread had claimed ownership to an MC bank, this technique will allow each logical processors to handle corrected MC errors independently and requires no synchronization to access shared MSR resources. Consult Example 17-5 for guidelines on logging when processing CMCI.

17.6 RECOVERY OF UNCORRECTED RECOVERABLE (UCR) ERRORS

Recovery of uncorrected recoverable machine check errors is an enhancement in machine-check architecture. The first processor that supports this feature is 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_2EH; see the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. This allows system software to perform recovery action on a certain class of uncorrected errors and continue execution.

17.6.1 Detection of Software Error Recovery Support

Software must use bit 24 of IA32_MCG_CAP (MCG_SER_P) to detect the presence of software error recovery support (see Figure 17-2). When IA32_MCG_CAP[24] is set, this indicates that the processor supports software error recovery. When this bit is clear, this indicates that there is no support for error recovery from the processor and the primary responsibility of the machine check handler is logging the machine check error information and shutting down the system.

The new class of architectural MCA errors from which system software can attempt recovery is called Uncorrected Recoverable (UCR) Errors. UCR errors are uncorrected errors that have been detected and signaled but have not corrupted the processor context. For certain UCR errors, this means that once system software has performed a certain recovery action, it is possible to continue execution on this processor. UCR error reporting provides an error containment mechanism for data poisoning. The machine check handler will use the error log information from the error reporting registers to analyze and implement specific error recovery actions for UCR errors.

17.6.2 UCR Error Reporting and Logging

IA32_MCi_STATUS MSR is used for reporting UCR errors and existing corrected or uncorrected errors. The definitions of IA32_MCi_STATUS, including bit fields to identify UCR errors, is shown in Figure 17-6. UCR errors can be signaled through either the corrected machine check interrupt (CMCI) or machine check exception (MCE) path depending on the type of the UCR error.

When IA32_MCG_CAP[24] is set, a UCR error is indicated by the following bit settings in the IA32_MCi_STATUS register:

- Valid (bit 63) = 1
- UC (bit 61) = 1
- PCC (bit 57) = 0

Additional information from the IA32_MCi_MISC and the IA32_MCi_ADDR registers for the UCR error are available when the ADDRV and the MISCV flags in the IA32_MCi_STATUS register are set (see Section 17.3.2.4). The MCA error code field of the IA32_MCi_STATUS register indicates the type of UCR error. System software can interpret the MCA error code field to analyze and identify the necessary recovery action for the given UCR error.

In addition, the IA32_MCi_STATUS register bit fields, bits 56:55, are defined (see Figure 17-6) to provide additional information to help system software to properly identify the necessary recovery action for the UCR error:

- S (Signaling) flag, bit 56 - Indicates (when set) that a machine check exception was generated for the UCR error reported in this MC bank and system software needs to check the AR flag and the MCA error code fields in the IA32_MCi_STATUS register to identify the necessary recovery action for this error. When the S flag in the IA32_MCi_STATUS register is clear, this UCR error was not signaled via a machine check exception and instead was reported as a corrected machine check (CMC). System software is not required to take any recovery action when the S flag in the IA32_MCi_STATUS register is clear.
- AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. This recovery action must be completed successfully before any additional work is scheduled for this processor. When the RIPV flag in the IA32_MCG_STATUS is clear, an alternative execution stream needs to be provided; when the MCA error code specific recovery specific recovery action cannot be successfully completed, system software must shut down the system. When the AR flag in the IA32_MCi_STATUS register is clear, system software may still take MCA error code specific recovery action but this is optional; system software can safely resume program execution at the instruction pointer saved on the stack from the machine check exception when the RIPV flag in the IA32_MCG_STATUS register is set.

Both the S and the AR flags in the IA32_MCi_STATUS register are defined to be sticky bits, which mean that once set, the processor does not clear them. Only software and good power-on reset can clear the S and the AR-flags. Both the S and the AR flags are only set when the processor reports the UCR errors (MCG_CAP[24] is set).

17.6.3 UCR Error Classification

With the S and AR flag encoding in the IA32_MCi_STATUS register, UCR errors can be classified as:

- Uncorrected no action required (UCNA) - is a UCR error that is not signaled via a machine check exception and, instead, is reported to system software as a corrected machine check error. UCNA errors indicate that some data in the system is corrupted, but the data has not been consumed and the processor state is valid and you may continue execution on this processor. UCNA errors require no action from system software to continue execution. A UCNA error is indicated with UC=1, PCC=0, S=0 and AR=0 in the IA32_MCi_STATUS register.
- Software recoverable action optional (SRAO) - a UCR error is signaled either via a machine check exception or CMCI. System software recovery action is optional and not required to continue execution from this machine check exception. SRAO errors indicate that some data in the system is corrupt, but the data has not been consumed and the processor state is valid. SRAO errors provide the additional error information for system software to perform a recovery action. An SRAO error when signaled as a machine check is indicated with UC=1, PCC=0, S=1, EN=1 and AR=0 in the IA32_MC*i*_STATUS register. In cases when SRAO is signaled via CMCI the error signature is indicated via UC=1, PCC=0, S=0. Recovery actions for SRAO errors are MCA error code specific. The MISCV and the ADDRV flags in the IA32_MC*i*_STATUS register are set when the additional error information is available from the IA32_MC*i*_MISC and the IA32_MC*i*_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MC*i*_STATUS register to identify the specific recovery action for a given SRAO error. If MISCV and ADDRV are not set, it is recommended that no system software error recovery be performed however, system software can resume execution.
- Software recoverable action required (SRAR) - a UCR error that requires system software to take a recovery action on this processor before scheduling another stream of execution on this processor. SRAR errors indicate

that the error was detected and raised at the point of the consumption in the execution flow. An SRAR error is indicated with UC=1, PCC=0, S=1, EN=1 and AR=1 in the IA32_MCi_STATUS register. Recovery actions are MCA error code specific. The MISCV and the ADDRV flags in the IA32_MCi_STATUS register are set when the additional error information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MCi_STATUS register to identify the specific recovery action for a given SRAR error. If MISCV and ADDRV are not set, it is recommended that system software shutdown the system.

Table 17-7 summarizes UCR, corrected, and uncorrected errors.

Table 17-7. MC Error Classifications

Type of Error[1]	UC	EN	PCC	S	AR	Signaling	Software Action	Example
Uncorrected Error (UC)	1	1	1	x	x	MCE	If EN=1, reset the system, else log and OK to keep the system running.	
SRAR	1	1	0	1	1	MCE	For known MCACOD, take specific recovery action; For unknown MCACOD, must bugcheck. If OVER=1, reset system, else take specific recovery action.	Cache to processor load error.
SRAO	1	x[2]	0	x[2]	0	MCE/CMC	For known MCACOD, take specific recovery action; For unknown MCACOD, OK to keep the system running.	Patrol scrub and explicit writeback poison errors.
UCNA	1	x	0	0	0	CMC	Log the error and Ok to keep the system running.	Poison detection error.
Corrected Error (CE)	0	x	x	x	x	CMC	Log the error and no corrective action required.	ECC in caches and memory.

NOTES:

1. SRAR, SRAO and UCNA errors are supported by the processor only when IA32_MCG_CAP[24] (MCG_SER_P) is set.
2. EN=1, S=1 when signaled via MCE. EN=x, S=0 when signaled via CMC.

17.6.4 UCR Error Overwrite Rules

In general, the overwrite rules are as follows:

- UCR errors will overwrite corrected errors.
- Uncorrected (PCC=1) errors overwrite UCR (PCC=0) errors.
- UCR errors are not written over previous UCR errors.
- Corrected errors do not write over previous UCR errors.

Regardless of whether the 1st error is retained or the 2nd error is overwritten over the 1st error, the OVER flag in the IA32_MCi_STATUS register will be set to indicate an overflow condition. As the S flag and AR flag in the IA32_MCi_STATUS register are defined to be sticky flags, a second event cannot clear these 2 flags once set, however the MC bank information may be filled in for the 2nd error. The table below shows the overwrite rules and how to treat a second error if the first event is already logged in a MC bank along with the resulting bit setting of the UC, PCC, and AR flags in the IA32_MCi_STATUS register. As UCNA and SRA0 errors do not require recovery action from system software to continue program execution, a system reset by system software is not required unless the AR flag or PCC flag is set for the UCR overflow case (OVER=1, VAL=1, UC=1, PCC=0).

Table 17-8 lists overwrite rules for uncorrected errors, corrected errors, and uncorrected recoverable errors.

Table 17-8. Overwrite Rules for UC, CE, and UCR Errors

First Event	Second Event	UC	PCC	S	AR	MCA Bank	Reset System
CE	UCR	1	0	0 if UCNA, else 1	1 if SRAR, else 0	second	yes, if AR=1
UCR	CE	1	0	0 if UCNA, else 1	1 if SRAR, else 0	first	yes, if AR=1

Table 17-8. Overwrite Rules for UC, CE, and UCR Errors

First Event	Second Event	UC	PCC	S	AR	MCA Bank	Reset System
UCNA	UCNA	1	0	0	0	first	no
UCNA	SRAO	1	0	1	0	first	no
UCNA	SRAR	1	0	1	1	first	yes
SRAO	UCNA	1	0	1	0	first	no
SRAO	SRAO	1	0	1	0	first	no
SRAO	SRAR	1	0	1	1	first	yes
SRAR	UCNA	1	0	1	1	first	yes
SRAR	SRAO	1	0	1	1	first	yes
SRAR	SRAR	1	0	1	1	first	yes
UCR	UC	1	1	undefined	undefined	second	yes
UC	UCR	1	1	undefined	undefined	first	yes

17.7 MACHINE-CHECK AVAILABILITY

The machine-check architecture and machine-check exception (#MC) are model-specific features. Software can execute the CPUID instruction to determine whether a processor implements these features. Following the execution of the CPUID instruction, the settings of the MCA flag (bit 14) and MCE flag (bit 7) in EDX indicate whether the processor implements the machine-check architecture and machine-check exception.

17.8 MACHINE-CHECK INITIALIZATION

To use the processors machine-check architecture, software must initialize the processor to activate the machine-check exception and the error-reporting mechanism.

Example 17-1 gives pseudocode for performing this initialization. This pseudocode checks for the existence of the machine-check architecture and exception; it then enables machine-check exception and the error-reporting register banks. The pseudocode shown is compatible with the Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors.

Following power up or power cycling, IA32_MC*i*_STATUS registers are not guaranteed to have valid data until after they are initially cleared to zero by software (as shown in the initialization pseudocode in Example 17-1).

Example 17-1. Machine-Check Initialization Pseudocode

```
Check CPUID Feature Flags for MCE and MCA support
IF CPU supports MCE
THEN
    IF CPU supports MCA
    THEN
        IF (IA32_MCG_CAP.MCG_CTL_P = 1)
        (* IA32_MCG_CTL register is present *)
        THEN
            IA32_MCG_CTL ← FFFFFFFFFFFFFFFFH;
            (* enables all MCA features *)
        FI

        IF (IA32_MCG_CAP.MCG_LMCE_P = 1 and IA32_FEATURE_CONTROL.LOCK = 1 and IA32_FEATURE_CONTROL.LMCE_ENABLED = 1)
        (* IA32_MCG_EXT_CTL register is present and platform has enabled LMCE to permit system software to use LMCE *)
        THEN
            IA32_MCG_EXT_CTL ← IA32_MCG_EXT_CTL | 01H;
            (* System software enables LMCE capability for hardware to signal MCE to a single logical processor*)
        FI
        (* Determine number of error-reporting banks supported *)
        COUNT← IA32_MCG_CAP.Count;
        MAX_BANK_NUMBER ← COUNT - 1;

        IF (Processor Family is 6H and Processor EXTMODEL:MODEL is less than 1AH)
        THEN
            (* Enable logging of all errors except for MC0_CTL register *)
            FOR error-reporting banks (1 through MAX_BANK_NUMBER)
            DO
                IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFFH;
            OD

        ELSE
            (* Enable logging of all errors including MC0_CTL register *)
            FOR error-reporting banks (0 through MAX_BANK_NUMBER)
            DO
                IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFFH;
            OD
        FI

        (* BIOS clears all errors only on power-on reset *)
        IF (BIOS detects Power-on reset)
        THEN
            FOR error-reporting banks (0 through MAX_BANK_NUMBER)
            DO
                IA32_MCi_STATUS ← 0;
            OD
        ELSE
            FOR error-reporting banks (0 through MAX_BANK_NUMBER)
            DO
                (Optional for BIOS and OS) Log valid errors
                (OS only) IA32_MCi_STATUS ← 0;
            OD

        FI
    FI

    Setup the Machine Check Exception (#MC) handler for vector 18 in IDT

    Set the MCE bit (bit 6) in CR4 register to enable Machine-Check Exceptions
FI
```

17.9 INTERPRETING THE MCA ERROR CODES

When the processor detects a machine-check error condition, it writes a 16-bit error code to the MCA error code field of one of the IA32_MC*i*_STATUS registers and sets the VAL (valid) flag in that register. The processor may also write a 16-bit model-specific error code in the IA32_MC*i*_STATUS register depending on the implementation of the machine-check architecture of the processor.

The MCA error codes are architecturally defined for Intel 64 and IA-32 processors. To determine the cause of a machine-check exception, the machine-check exception handler must read the VAL flag for each IA32_MC*i*_STATUS register. If the flag is set, the machine check-exception handler must then read the MCA error code field of the register. It is the encoding of the MCA error code field [15:0] that determines the type of error being reported and not the register bank reporting it.

There are two types of MCA error codes: simple error codes and compound error codes.

17.9.1 Simple Error Codes

Table 17-9 shows the simple error codes. These unique codes indicate global error information.

Table 17-9. IA32_MCi_Status [15:0] Simple Error Code Encoding

Error Code	Binary Encoding	Meaning
No Error	0000 0000 0000 0000	No error has been reported to this bank of error-reporting registers.
Unclassified	0000 0000 0000 0001	This error has not been classified into the MCA error classes.
Microcode ROM Parity Error	0000 0000 0000 0010	Parity error in internal microcode ROM
External Error	0000 0000 0000 0011	The BINIT# from another processor caused this processor to enter machine check.[1]
FRC Error	0000 0000 0000 0100	FRC (functional redundancy check) main/secondary error.
Internal Parity Error	0000 0000 0000 0101	Internal parity error.
SMM Handler Code Access Violation	0000 0000 0000 0110	An attempt was made by the SMM Handler to execute outside the ranges specified by SMRR.
Internal Timer Error	0000 0100 0000 0000	Internal timer error.
I/O Error	0000 1110 0000 1011	generic I/O error.
Internal Unclassified	0000 01xx xxxx xxxx	Internal unclassified errors. [2]

NOTES:

1. BINIT# assertion will cause a machine check exception if the processor (or any processor on the same external bus) has BINIT# observation enabled during power-on configuration (hardware strapping) and if machine check exceptions are enabled (by setting CR4.MCE = 1).
2. At least one X must equal one. Internal unclassified errors have not been classified.

17.9.2 Compound Error Codes

Compound error codes describe errors related to the TLBs, memory, caches, bus and interconnect logic, and internal timer. A set of sub-fields is common to all of compound errors. These sub-fields describe the type of access, level in the cache hierarchy, and type of request. Table 17-10 shows the general form of the compound error codes.

Table 17-10. IA32_MCi_Status [15:0] Compound Error Code Encoding

Type	Form	Interpretation
Generic Cache Hierarchy	000F 0000 0000 11LL	Generic cache hierarchy error
TLB Errors	000F 0000 0001 TTLL	{TT}TLB{LL}_ERR
Memory Controller Errors	000F 0000 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Cache Hierarchy Errors	000F 0001 RRRR TTLL	{TT}CACHE{LL}_{RRRR}_ERR
Extended Memory Errors	000F 0010 1MMM CCCC	{MMM}_CHANNEL{CCCC}_ERR
Bus and Interconnect Errors	000F 1PPT RRRR IILL	BUS{LL}_{PP}_{RRRR}_{II}_{T}_ERR

The "Interpretation" column in the table indicates the name of a compound error. The name is constructed by substituting mnemonics for the sub-field names given within curly braces. For example, the error code ICACHEL1_RD_ERR is constructed from the form:

{TT}CACHE{LL}_{RRRR}_ERR,
where {TT} is replaced by I, {LL} is replaced by L1, and {RRRR} is replaced by RD.

For more information on the "Form" and "Interpretation" columns, see Section 17.9.2.1, "Correction Report Filtering (F) Bit," through Section 17.9.2.5, "Bus and Interconnect Errors."

17.9.2.1 Correction Report Filtering (F) Bit

Starting with Intel Core Duo processors, bit 12 in the "Form" column in Table 17-10 is used to indicate that a particular posting to a log may be the last posting for corrections in that line/entry, at least for some time:

- 0 in bit 12 indicates “normal” filtering (original P6/Pentium4/Atom/Xeon processor meaning).
- 1 in bit 12 indicates “corrected” filtering (filtering is activated for the line/entry in the posting). Filtering means that some or all of the subsequent corrections to this entry (in this structure) will not be posted. The enhanced error reporting introduced with the Intel Core Duo processors is based on tracking the lines affected by repeated corrections (see Section 17.4, “Enhanced Cache Error reporting”). This capability is indicated by IA32_MCG_CAP[11]. Only the first few correction events for a line are posted; subsequent redundant correction events to the same line are not posted. Uncorrected events are always posted.

The behavior of error filtering after crossing the yellow threshold is model-specific. Filtering has meaning only for corrected errors (UC=0 in IA32_MCi_STATUS MSR). System software must ignore filtering bit (12) for uncorrected errors.

17.9.2.2 Transaction Type (TT) Sub-Field

The 2-bit TT sub-field (Table 17-11) indicates the type of transaction (data, instruction, or generic). The sub-field applies to the TLB, cache, and interconnect error conditions. Note that interconnect error conditions are primarily associated with P6 family and Pentium processors, which utilize an external APIC bus separate from the system bus. The generic type is reported when the processor cannot determine the transaction type.

Table 17-11. Encoding for TT (Transaction Type) Sub-Field

Transaction Type	Mnemonic	Binary Encoding
Instruction	I	00
Data	D	01
Generic	G	10

17.9.2.3 Level (LL) Sub-Field

The 2-bit LL sub-field (see Table 17-12) indicates the level in the memory hierarchy where the error occurred (level 0, level 1, level 2, or generic). The LL sub-field also applies to the TLB, cache, and interconnect error conditions. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support two levels in the cache hierarchy and one level in the TLBs. Again, the generic type is reported when the processor cannot determine the hierarchy level.

Table 17-12. Level Encoding for LL (Memory Hierarchy Level) Sub-Field

Hierarchy Level	Mnemonic	Binary Encoding
Level 0	L0	00
Level 1	L1	01
Level 2	L2	10
Generic	LG	11

17.9.2.4 Request (RRRR) Sub-Field

The 4-bit RRRR sub-field (see Table 17-13) indicates the type of action associated with the error. Actions include read and write operations, prefetches, cache evictions, and snoops. Generic error is returned when the type of error cannot be determined. Generic read and generic write are returned when the processor cannot determine the type of instruction or data request that caused the error. Eviction and snoop requests apply only to the caches. All of the other requests apply to TLBs, caches, and interconnects.

Table 17-13. Encoding of Request (RRRR) Sub-Field

Request Type	Mnemonic	Binary Encoding
Generic Error	ERR	0000
Generic Read	RD	0001
Generic Write	WR	0010
Data Read	DRD	0011
Data Write	DWR	0100
Instruction Fetch	IRD	0101
Prefetch	PREFETCH	0110
Eviction	EVICT	0111
Snoop	SNOOP	1000
Page Walk	PW	1001
EPT Page Walk	EPW	1010

17.9.2.5 Bus and Interconnect Errors

The bus and interconnect errors are defined with the 2-bit PP (participation), 1-bit T (time-out), and 2-bit II (memory or I/O) sub-fields, in addition to the LL and RRRR sub-fields (see Table 17-14). The bus error conditions are implementation dependent and related to the type of bus implemented by the processor. Likewise, the interconnect error conditions are predicated on a specific implementation-dependent interconnect model that describes the connections between the different levels of the storage hierarchy. The type of bus is implementation dependent, and as such is not specified in this document. A bus or interconnect transaction consists of a request involving an address and a response.

Table 17-14. Encodings of PP, T, and II Sub-Fields

Sub-Field	Transaction	Mnemonic	Binary Encoding
PP (Participation)	Local processor* originated request	SRC	00
	Local processor* responded to request	RES	01
	Local processor* observed error as third party	OBS	10
	Generic		11
T (Time-out)	Request timed out	TIMEOUT	1
	Request did not time out	NOTIMEOUT	0
II (Memory or I/O)	Memory Access	M	00
	Reserved		01
	I/O	IO	10
	Other transaction		11

NOTE:

* Local processor differentiates the processor reporting the error from other system components (including the APIC, other processors, etc.).

17.9.2.6 Memory Controller and Extended Memory Errors

The memory controller errors are defined with the 3-bit MMM (memory transaction type), and 4-bit CCCC (channel) sub-fields. The encodings for MMM and CCCC are defined in Table 17-15. Extended Memory errors use the same encodings and are used to report errors in memory used as a cache.

Table 17-15. Encodings of MMM and CCCC Sub-Fields

Sub-Field	Transaction	Mnemonic	Binary Encoding
MMM	Generic undefined request	GEN	000
	Memory read error	RD	001
	Memory write error	WR	010
	Address/Command Error	AC	011
	Memory Scrubbing Error	MS	100
	Reserved		101-111
CCCC	Channel number	CHN	0000-1110
	Channel not specified		1111

Note that the CCCC channel number may be enumerated from zero separately by each memory controller on a system. On a multi-socket system, or a system with multiple memory controllers per socket, it is necessary to also consider which machine check bank logged the error. See Chapter 18 for details on specific implementations.

17.9.3 Architecturally Defined UCR Errors

Software recoverable compound error code are defined in this section.

17.9.3.1 Architecturally Defined SRAO Errors

The following two SRAO errors are architecturally defined.

- UCR Errors detected by memory controller scrubbing; and
- UCR Errors detected during L3 cache (L3) explicit writebacks.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 17-10). Their values and compound encoding format are given in Table 17-16.

Table 17-16. MCA Compound Error Code Encoding for SRAO Errors

Type	MCACOD Value	MCA Error Code Encoding[1]
Memory Scrubbing	C0H - CFH	0000_0000_1100_CCCC 000F 0000 1MMM CCCC (Memory Controller Error), where Memory subfield MMM = 100B (memory scrubbing) Channel subfield CCCC = channel # or generic
L3 Explicit Writeback	17AH	0000_0001_0111_1010 000F 0001 RRRR TTLL (Cache Hierarchy Error) where Request subfields RRRR = 0111B (Eviction) Transaction Type subfields TT = 10B (Generic) Level subfields LL = 10B

NOTES:

1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 17-17 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAO errors.

Table 17-17. IA32_MCi_STATUS Values for SRAO Errors

SRAO Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR	MCACOD
Memory Scrubbing	1	0	1	x[1]	1	1	0	x[1]	0	C0H-CFH
L3 Explicit Writeback	1	0	1	x[1]	1	1	0	x[1]	0	17AH

NOTES:

1. When signaled as MCE, EN=1 and S=1. If error was signaled via CMC, then EN=x, and S=0.

For both the memory scrubbing and L3 explicit writeback errors, the ADDRV and MISCV flags in the IA32_MCi_STATUS register are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the memory scrubbing and L3 explicit writeback errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors as outlined in Section 17.10.4.1. If LMCE is supported and enabled, some errors (not limited to UCR errors) may be delivered to only a single logical processor. System software should consult IA32_MCG_STATUS.LMCE_S to determine if the MCE signaled is only to this logical processor.

IA32_MCi_STATUS banks can be shared by logical processors within a core or within the same package. So several logical processors may find an SRAO error in the shared IA32_MCi_STATUS bank but other processors do not find it in any of the IA32_MCi_STATUS banks. Table 17-18 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the memory scrubbing and L3 explicit writeback errors on both the reporting and non-reporting logical processors.

Table 17-18. IA32_MCG_STATUS Flag Indication for SRAO Errors

SRAO Type	**Reporting Logical Processors**		**Non-reporting Logical Processors**	
	RIPV	**EIPV**	**RIPV**	**EIPV**
Memory Scrubbing	1	0	1	0
L3 Explicit Writeback	1	0	1	0

17.9.3.2 Architecturally Defined SRAR Errors

The following six SRAR errors are architecturally defined:

- UCR Errors detected on data load;
- UCR Errors detected on data page walk;
- UCR Errors detected on data page walk on EPT;
- UCR Errors detected on instruction fetch;
- UCR Errors detected on instruction fetch page walk; and
- UCR Errors detected on instruction fetch page walk on EPT.

The MCA error code encodings for these six architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 17-10). Their values and compound encoding format are given in Table 17-19.

Table 17-19. MCA Compound Error Code Encoding for SRAR Errors

Type	MCACOD Value	MCA Error Code Encoding[1]
Data Load	134H	0000_0001_0011_0100: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0011B (Data Load), Transaction Type subfield TT= 01B (Data), and Level subfield LL = 00B (Level 0).
Data Page Walk	194H	0000_0001_1001_0100: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1001B (Page Walk), Transaction Type subfield TT= 01B (Data), and Level subfield LL = 00B (Level 0).
Data Page Walk on EPT	1A4H	0000_0001_1010_0100: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1010B (EPT Page Walk), Transaction Type subfield TT= 01B (Data), and Level subfield LL = 00B (Level 0).
Instruction Fetch	150H	0000_0001_0101_0000: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 0101B (Instruction Fetch), Transaction Type subfield TT= 00B (Instruction), and Level subfield LL = 00B (Level 0).
Instruction Fetch Page Walk	190H	0000_0001_1001_0000: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1001B (Page Walk), Transaction Type subfield TT= 00B (Instruction), and Level subfield LL = 00B (Level 0).
Instruction Fetch Page Walk on EPT	1A0H	0000_0001_1010_0000: 000F 0001 RRRR TTLL (Cache Hierarchy Error), where Request subfield RRRR = 1010B (EPT Page Walk), Transaction Type subfield TT= 00B (Instruction), and Level subfield LL = 00B (Level 0).

NOTES:

1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 17-20 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAR errors.

Table 17-20. IA32_MCi_STATUS Values for All Defined SRAR Errors

SRAR Error	Valid	OVER	UC	EN	MISCV	ADDRV	PCC	S	AR
All defined SRAR errors defined in Table 17-19	1	0	1	1	1	1	0	1	1

For all defined SRAR errors, the ADDRV and MISCV flags in the IA32_MCi_STATUS register are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the data load and instruction fetch errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors on the system on which the UCR errors are supported, except when the processor supports LMCE and LMCE is enabled by system software (see Section 17.3.1.5). The IA32_MCG_STATUS MSR allows system software to distinguish the affected logical processor of an SRAR error amongst logical processors that observed SRAR via MCi_STATUS bank.

Table 17-21 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the data load and instruction fetch errors on both the reporting and non-reporting logical processors. The recoverable SRAR error reported by a processor may be continuable, where the system software can interpret the context of continuable as follows: the error was isolated, contained. If software can rectify the error condition in the current instruction stream, the execution context on that logical processor can be continued without loss of information.

Table 17-21. IA32_MCG_STATUS Flag Indication for SRAR Errors

SRAR Type	Affected Logical Processor			Non-Affected Logical Processors		
	RIPV	**EIPV**	**Continuable**	**RIPV**	**EIPV**	**Continuable**
Recoverable-continuable	1	1	Yes[1]	1	0	Yes
Recoverable-not-continuable	0	x	No			

NOTES:

1. See the definition of the context of “continuable” above and additional details below.

SRAR Error And Affected Logical Processors

The affected logical processor is the one that has detected and raised an SRAR error at the point of the consumption in the execution flow. The affected logical processor should find the Data Load or the Instruction Fetch error information in the IA32_MCi_STATUS register that is reporting the SRAR error.

Table 17-21 list the actionable scenarios that system software can respond to an SRAR error on an affected logical processor according to RIPV and EIPV values:

- Recoverable-continuable SRAR Error (RIPV=1, EIPV=1):

  For recoverable-continuable SRAR errors, the affected logical processor should find that both the IA32_MCG_STATUS.RIPV and the IA32_MCG_STATUS.EIPV flags are set, indicating that system software may be able to restart execution from the interrupted context if it is able to rectify the error condition. If system software cannot rectify the error condition then it must treat the error as a recoverable error where restarting execution with the interrupted context is not possible. Restarting without rectifying the error condition will result in most cases with another SRAR error on the same instruction.

- Recoverable-not-continuable SRAR Error (RIPV=0, EIPV=x):

  For recoverable-not-continuable errors, the affected logical processor should find that either

  — IA32_MCG_STATUS.RIPV= 0, IA32_MCG_STATUS.EIPV=1, or

  — IA32_MCG_STATUS.RIPV= 0, IA32_MCG_STATUS.EIPV=0.

  In either case, this indicates that the error is detected at the instruction pointer saved on the stack for this machine check exception and restarting execution with the interrupted context is not possible. System software may take the following recovery actions for the affected logical processor:

  - The current executing thread cannot be continued. System software must terminate the interrupted stream of execution and provide a new stream of execution on return from the machine check handler for the affected logical processor.

SRAR Error And Non-Affected Logical Processors

The logical processors that observed but not affected by an SRAR error should find that the RIPV flag in the IA32_MCG_STATUS register is set and the EIPV flag in the IA32_MCG_STATUS register is cleared, indicating that it is safe to restart the execution at the instruction saved on the stack for the machine check exception on these processors after the recovery action is successfully taken by system software.

17.9.4 Multiple MCA Errors

When multiple MCA errors are detected within a certain detection window, the processor may aggregate the reporting of these errors together as a single event, i.e., a single machine exception condition. If this occurs, system software may find multiple MCA errors logged in different MC banks on one logical processor or find multiple MCA errors logged across different processors for a single machine check broadcast event. In order to handle multiple UCR errors reported from a single machine check event and possibly recover from multiple errors, system software may consider the following:

- Whether it can recover from multiple errors is determined by the most severe error reported on the system. If the most severe error is found to be an unrecoverable error (VAL=1, UC=1, PCC=1 and EN=1) after system software examines the MC banks of all processors to which the MCA signal is broadcast, recovery from the multiple errors is not possible and system software needs to reset the system.

- When multiple recoverable errors are reported and no other fatal condition (e.g., overflowed condition for SRAR error) is found for the reported recoverable errors, it is possible for system software to recover from the multiple recoverable errors by taking necessary recovery action for each individual recoverable error. However, system software can no longer expect one to one relationship with the error information recorded in the IA32_MCi_STATUS register and the states of the RIPV and EIPV flags in the IA32_MCG_STATUS register as the states of the RIPV and the EIPV flags in the IA32_MCG_STATUS register may indicate the information for the most severe error recorded on the processor. System software is required to use the RIPV flag indication in the IA32_MCG_STATUS register to make a final decision of recoverability of the errors and find the restart-ability requirement after examining each IA32_MCi_STATUS register error information in the MC banks.

  In certain cases where system software observes more than one SRAR error logged for a single logical processor, it can no longer rely on affected threads as specified in Table 15-20 above. System software is recommended to reset the system if this condition is observed.

17.9.5 Machine-Check Error Codes Interpretation

Chapter 18, "Interpreting Machine Check Error Codes," provides information on interpreting the MCA error code, model-specific error code, and other information error code fields. For P6 family processors, information has been included on decoding external bus errors. For Pentium 4 and Intel Xeon processors; information is included on external bus, internal timer and cache hierarchy errors.

17.10 GUIDELINES FOR WRITING MACHINE-CHECK SOFTWARE

The machine-check architecture and error logging can be used in three different ways:

- To detect machine errors during normal instruction execution, using the machine-check exception (#MC).
- To periodically check and log machine errors.
- To examine recoverable UCR errors, determine software recoverability and perform recovery actions via a machine-check exception handler or a corrected machine-check interrupt handler.

To use the machine-check exception, the operating system or executive software must provide a machine-check exception handler. This handler may need to be designed specifically for each family of processors.

A special program or utility is required to log machine errors.

Guidelines for writing a machine-check exception handler or a machine-error logging utility are given in the following sections.

17.10.1 Machine-Check Exception Handler

The machine-check exception (#MC) corresponds to vector 18. To service machine-check exceptions, a trap gate must be added to the IDT. The pointer in the trap gate must point to a machine-check exception handler. Two approaches can be taken to designing the exception handler:

1. The handler can merely log all the machine status and error information, then call a debugger or shut down the system.
2. The handler can analyze the reported error information and, in some cases, attempt to correct the error and restart the processor.

For Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors; virtually all machine-check conditions cannot be corrected (they result in abort-type exceptions). The logging of status and error information is therefore a baseline implementation requirement.

When IA32_MCG_CAP[24] is clear, consider the following when writing a machine-check exception handler:

- To determine the nature of the error, the handler must read each of the error-reporting register banks. The count field in the IA32_MCG_CAP register gives number of register banks. The first register of register bank 0 is at address 400H.

- The VAL (valid) flag in each IA32_MC*i*_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and do not need to be checked.
- To write a portable exception handler, only the MCA error code field in the IA32_MC*i*_STATUS register should be checked. See Section 17.9, "Interpreting the MCA Error Codes," for information that can be used to write an algorithm to interpret this field.
- Correctable errors are corrected automatically by the processor. The UC flag in each IA32_MCi_STATUS register indicates whether the processor automatically corrected an error.
- The RIPV, PCC, and OVER flags in each IA32_MCi_STATUS register indicate whether recovery from the error is possible. If PCC or OVER are set, recovery is not possible. If RIPV is not set, program execution can not be restarted reliably. When recovery is not possible, the handler typically records the error information and signals an abort to the operating system.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether the program can be restarted at the instruction indicated by the instruction pointer (the address of the instruction pushed on the stack when the exception was generated). If this flag is clear, the processor may still be able to be restarted (for debugging purposes) but not without loss of program continuity.
- For unrecoverable errors, the EIPV flag in the IA32_MCG_STATUS register indicates whether the instruction indicated by the instruction pointer pushed on the stack (when the exception was generated) is related to the error. If the flag is clear, the pushed instruction may not be related to the error.
- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. Before returning from the machine-check exception handler, software should clear this flag so that it can be used reliably by an error logging utility. The MCIP flag also detects recursion. The machine-check architecture does not support recursion. When the processor detects machine-check recursion, it enters the shutdown state.

Example 17-2 gives typical steps carried out by a machine-check exception handler.

Example 17-2. Machine-Check Exception Handler Pseudocode

```
IF CPU supports MCE
    THEN
        IF CPU supports MCA
            THEN
                call errorlogging routine; (* returns restartability *)
        FI;
    ELSE (* Pentium(R) processor compatible *)
        READ P5_MC_ADDR
        READ P5_MC_TYPE;
        report RESTARTABILITY to console;
FI;
IF error is not restartable
    THEN
        report RESTARTABILITY to console;
        abort system;
FI;
CLEAR MCIP flag in IA32_MCG_STATUS;
```

17.10.2 Pentium Processor Machine-Check Exception Handling

Machine-check exception handler on P6 family, Intel Atom and later processor families, should follow the guidelines described in Section 17.10.1 and Example 17-2 that check the processor's support of MCA.

NOTE

> On processors that support MCA (CPUID.1.EDX.MCA = 1) reading the P5_MC_TYPE and P5_MC_ADDR registers may produce invalid data.

When machine-check exceptions are enabled for the Pentium processor (MCE flag is set in control register CR4), the machine-check exception handler uses the RDMSR instruction to read the error type from the P5_MC_TYPE register and the machine check address from the P5_MC_ADDR register. The handler then normally reports these register values to the system console before aborting execution (see Example 17-2).

17.10.3 Logging Correctable Machine-Check Errors

The error handling routine for servicing the machine-check exceptions is responsible for logging uncorrected errors.

If a machine-check error is correctable, the processor does not generate a machine-check exception for it. To detect correctable machine-check errors, a utility program must be written that reads each of the machine-check error-reporting register banks and logs the results in an accounting file or data structure. This utility can be implemented in either of the following ways.

- A system daemon that polls the register banks on an infrequent basis, such as hourly or daily.
- A user-initiated application that polls the register banks and records the exceptions. Here, the actual polling service is provided by an operating-system driver or through the system call interface.
- An interrupt service routine servicing CMCI can read the MC banks and log the error. Please refer to Section 17.10.4.2 for guidelines on logging correctable machine checks.

Example 17-3 gives pseudocode for an error logging utility.

Example 17-3. Machine-Check Error Logging Pseudocode

```
Assume that execution is restartable;
IF the processor supports MCA
    THEN
    FOR each bank of machine-check registers
        DO
            READ IA32_MCi_STATUS;
            IF VAL flag in IA32_MCi_STATUS = 1
                THEN
                    IF ADDRV flag in IA32_MCi_STATUS = 1
                        THEN READ IA32_MCi_ADDR;
                    FI;
                    IF MISCV flag in IA32_MCi_STATUS = 1
                        THEN READ IA32_MCi_MISC;
                    FI;
                    IF MCIP flag in IA32_MCG_STATUS = 1
                        (* Machine-check exception is in progress *)
                        AND PCC flag in IA32_MCi_STATUS = 1
                        OR RIPV flag in IA32_MCG_STATUS = 0
                        (* execution is not restartable *)
                            THEN
                                RESTARTABILITY = FALSE;
                                return RESTARTABILITY to calling procedure;
                    FI;
                    Save time-stamp counter and processor ID;
                    Set IA32_MCi_STATUS to all 0s;
                    Execute serializing instruction (i.e., CPUID);
            FI;
        OD;
FI;
```

If the processor supports the machine-check architecture, the utility reads through the banks of error-reporting registers looking for valid register entries. It then saves the values of the IA32_MC*i*_STATUS, IA32_MC*i*_ADDR, IA32_MC*i*_MISC, and IA32_MCG_STATUS registers for each bank that is valid. The routine minimizes processing time by recording the raw data into a system data structure or file, reducing the overhead associated with polling. User utilities analyze the collected data in an off-line environment.

When the MCIP flag is set in the IA32_MCG_STATUS register, a machine-check exception is in progress and the machine-check exception handler has called the exception logging routine.

Once the logging process has been completed the exception-handling routine must determine whether execution can be restarted, which is usually possible when damage has not occurred (The PCC flag is clear, in the IA32_MC*i*_STATUS register) and when the processor can guarantee that execution is restartable (the RIPV flag is set in the IA32_MCG_STATUS register). If execution cannot be restarted, the system is not recoverable and the exception-handling routine should signal the console appropriately before returning the error status to the Operating System kernel for subsequent shutdown.

The machine-check architecture allows buffering of exceptions from a given error-reporting bank although the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors do not implement this feature. The error logging routine should provide compatibility with future processors by reading each hardware error-reporting bank's IA32_MC*i*_STATUS register and then writing 0s to clear the OVER and VAL flags in this register. The error logging utility should re-read the IA32_MC*i*_STATUS register for the bank ensuring that the valid bit is clear. The processor will write the next error into the register bank and set the VAL flags.

Additional information that should be stored by the exception-logging routine includes the processor’s time-stamp counter value, which provides a mechanism to indicate the frequency of exceptions. A multiprocessing operating system stores the identity of the processor node incurring the exception using a unique identifier, such as the processor’s APIC ID (see Section 12.8, “Handling Interrupts”).

The basic algorithm given in Example 17-3 can be modified to provide more robust recovery techniques. For example, software has the flexibility to attempt recovery using information unavailable to the hardware. Specifically, the machine-check exception handler can, after logging carefully analyze the error-reporting registers when the error-logging routine reports an error that does not allow execution to be restarted. These recovery techniques can use external bus related model-specific information provided with the error report to localize the source of the error within the system and determine the appropriate recovery strategy.

17.10.4 Machine-Check Software Handler Guidelines for Error Recovery

17.10.4.1 Machine-Check Exception Handler for Error Recovery

When writing a machine-check exception (MCE) handler to support software recovery from Uncorrected Recoverable (UCR) errors, consider the following:

- When IA32_MCG_CAP [24] is zero, there are no recoverable errors supported and all machine-check are fatal exceptions. The logging of status and error information is therefore a baseline implementation requirement.
- When IA32_MCG_CAP [24] is 1, certain uncorrected errors called uncorrected recoverable (UCR) errors may be software recoverable. The handler can analyze the reported error information, and in some cases attempt to recover from the uncorrected error and continue execution.
- For processors on which CPUID reports DisplayFamily_DisplayModel as 06H_0EH and onward, an MCA signal is broadcast to all logical processors in the system; see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. Due to the potentially shared machine check MSR resources among the logical processors on the same package/core, the MCE handler may be required to synchronize with the other processors that received a machine check error and serialize access to the machine check registers when analyzing, logging, and clearing the information in the machine check registers.
    - On processors that indicate ability for local machine-check exception (MCG_LMCE_P), hardware can choose to report the error to only a single logical processor if system software has enabled LMCE by setting IA32_MCG_EXT_CTL[LMCE_EN] = 1 as outlined in Section 17.3.1.5.
- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and should not be checked.
- The MCE handler is primarily responsible for processing uncorrected errors. The UC flag in each IA32_MCi_Status register indicates whether the reported error was corrected (UC=0) or uncorrected (UC=1). The MCE handler can optionally log and clear the corrected errors in the MC banks if it can implement software algorithm to avoid the undesired race conditions with the CMCI or CMC polling handler.
- For uncorrectable errors, the EIPV flag in the IA32_MCG_STATUS register indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is

generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.

- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. When a machine check exception is generated, it is expected that the MCIP flag in the IA32_MCG_STATUS register is set to 1. If it is not set, this machine check was generated by either an INT 18 instruction or some piece of hardware signaling an interrupt with vector 18.

When IA32_MCG_CAP [24] is 1, the following rules can apply when writing a machine check exception (MCE) handler to support software recovery:

- The PCC flag in each IA32_MCi_STATUS register indicates whether recovery from the error is possible for uncorrected errors (UC=1). If the PCC flag is set for enabled uncorrected errors (UC=1 and EN=1), recovery is not possible. When recovery is not possible, the MCE handler typically records the error information and signals the operating system to reset the system.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether restarting the program execution from the instruction pointer saved on the stack for the machine check exception is possible. When the RIPV is set, program execution can be restarted reliably when recovery is possible. If the RIPV flag is not set, program execution cannot be restarted reliably. In this case the recovery algorithm may involve terminating the current program execution and resuming an alternate thread of execution upon return from the machine check handler when recovery is possible. When recovery is not possible, the MCE handler signals the operating system to reset the system.
- When the EN flag is zero but the VAL and UC flags are one in the IA32_MCi_STATUS register, the reported uncorrected error in this bank is not enabled. As uncorrected errors with the EN flag = 0 are not the source of machine check exceptions, the MCE handler should log and clear non-enabled errors when the S bit is set and should continue searching for enabled errors from the other IA32_MCi_STATUS registers. Note that when IA32_MCG_CAP [24] is 0, any uncorrected error condition (VAL =1 and UC=1) including the one with the EN flag cleared are fatal and the handler must signal the operating system to reset the system. For the errors that do not generate machine check exceptions, the EN flag has no meaning.
- When the VAL flag is one, the UC flag is one, the EN flag is one and the PCC flag is zero in the IA32_MCi_STATUS register, the error in this bank is an uncorrected recoverable (UCR) error. The MCE handler needs to examine the S flag and the AR flag to find the type of the UCR error for software recovery and determine if software error recovery is possible.
- When both the S and the AR flags are clear in the IA32_MCi_STATUS register for the UCR error (VAL=1, UC=1, EN=x and PCC=0), the error in this bank is an uncorrected no-action required error (UCNA). UCNA errors are uncorrected but do not require any OS recovery action to continue execution. These errors indicate that some data in the system is corrupt, but that data has not been consumed and may not be consumed. If that data is consumed a non-UCNA machine check exception will be generated. UCNA errors are signaled in the same way as corrected machine check errors and the CMCI and CMC polling handler is primarily responsible for handling UCNA errors. Like corrected errors, the MCA handler can optionally log and clear UCNA errors as long as it can avoid the undesired race condition with the CMCI or CMC polling handler. As UCNA errors are not the source of machine check exceptions, the MCA handler should continue searching for uncorrected or software recoverable errors in all other MC banks.
- When the S flag in the IA32_MCi_STATUS register is set for the UCR error ((VAL=1, UC=1, EN=1 and PCC=0), the error in this bank is software recoverable and it was signaled through a machine-check exception. The AR flag in the IA32_MCi_STATUS register further clarifies the type of the software recoverable errors.
- When the AR flag in the IA32_MCi_STATUS register is clear for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action optional (SRAO) error. The MCE handler and the operating system can analyze the IA32_MCi_STATUS [15:0] to implement MCA error code specific optional recovery action, but this recovery action is optional. System software can resume the program execution from the instruction pointer saved on the stack for the machine check exception when the RIPV flag in the IA32_MCG_STATUS register is set.
- Even if the OVER flag in the IA32_MCi_STATUS register is set for the SRAO error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=0), the MCE handler can take recovery action for the SRAO error logged in the IA32_MCi_STATUS register. Since the recovery action for SRAO errors is optional, restarting the program execution from the instruction pointer saved on the stack for the machine check exception is still possible for the overflowed SRAO error if the RIPV flag in the IA32_MCG_STATUS is set.

- When the AR flag in the IA32_MCi_STATUS register is set for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action required (SRAR) error. The MCE handler and the operating system must take recovery action in order to continue execution after the machine-check exception. The MCA handler and the operating system need to analyze the IA32_MCi_STATUS [15:0] to determine the MCA error code specific recovery action. If no recovery action can be performed, the operating system must reset the system.
- When the OVER flag in the IA32_MCi_STATUS register is set for the SRAR error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=1), the MCE handler cannot take recovery action as the information of the SRAR error in the IA32_MCi_STATUS register was potentially lost due to the overflow condition. Since the recovery action for SRAR errors must be taken, the MCE handler must signal the operating system to reset the system.
- When the MCE handler cannot find any uncorrected (VAL=1, UC=1 and EN=1) or any software recoverable errors (VAL=1, UC=1, EN=1, PCC=0 and S=1) in any of the IA32_MCi banks of the processors, this is an unexpected condition for the MCE handler and the handler should signal the operating system to reset the system.
- Before returning from the machine-check exception handler, software must clear the MCIP flag in the IA32_MCG_STATUS register. The MCIP flag is used to detect recursion. The machine-check architecture does not support recursion. When the processor receives a machine check when MCIP is set, it automatically enters the shutdown state.

Example 17-4 gives pseudocode for an MC exception handler that supports recovery of UCR.

Example 17-4. Machine-Check Error Handler Pseudocode Supporting UCR

```
MACHINE CHECK HANDLER:  (* Called from INT 18 handler *)
NOERROR = TRUE;
ProcessorCount = 0;
IF CPU supports MCA
    THEN
        RESTARTABILITY = TRUE;
        IF (Processor Family = 6 AND DisplayModel ≥ 0EH) OR (Processor Family > 6)
            THEN
                IF ( MCG_LMCE = 1)
                    MCA_BROADCAST = FALSE;
                ELSE
                    MCA_BROADCAST = TRUE;
                FI;
                Acquire SpinLock;
                ProcessorCount++;  (* Allowing one logical processor at a time to examine machine check registers *)
                CALL MCA ERROR PROCESSING; (* returns RESTARTABILITY and NOERROR *)
            ELSE
                MCA_BROADCAST = FALSE;
                (* Implement a rendezvous mechanism with the other processors if necessary *)
                CALL MCA ERROR PROCESSING;
        FI;
    ELSE (* Pentium(R) processor compatible *)
        READ P5_MC_ADDR
        READ P5_MC_TYPE;
        RESTARTABILITY = FALSE;
FI;

IF NOERROR = TRUE
    THEN
        IF NOT (MCG_RIPV = 1 AND MCG_EIPV = 0)
            THEN
                RESTARTABILITY = FALSE;
        FI
FI;

IF RESTARTABILITY = FALSE
    THEN
        Report RESTARTABILITY to console;
        Reset system;
```

```
FI;

IF MCA_BROADCAST = TRUE
    THEN
        IF ProcessorCount = MAX_PROCESSORS
          AND NOERROR = TRUE
            THEN
                Report RESTARTABILITY to console;
                Reset system;
        FI;
        Release SpinLock;
        Wait till ProcessorCount = MAX_PROCESSRS on system;
        (* implement a timeout and abort function if necessary *)
FI;
CLEAR IA32_MCG_STATUS;
RESUME Execution;
(* End of MACHINE CHECK HANDLER*)

MCA ERROR PROCESSING:    (* MCA Error Processing Routine called from MCA Handler *)
IF MCIP flag in IA32_MCG_STATUS = 0
    THEN (* MCIP=0 upon MCA is unexpected *)
        RESTARTABILITY = FALSE;
FI;
FOR each bank of machine-check registers
    DO
        CLEAR_MC_BANK = FALSE;
        READ IA32_MCi_STATUS;
        IF VAL Flag in IA32_MCi_STATUS = 1
            THEN
                IF UC Flag in IA32_MCi_STATUS = 1
                    THEN
                        IF Bit 24 in IA32_MCG_CAP = 0
                            THEN (* the processor does not support software error recovery *)
                                RESTARTABILITY = FALSE;
                                NOERROR = FALSE;
                                GOTO LOG MCA REGISTER;
                        FI;
                        (* the processor supports software error recovery *)
                        IF EN Flag in IA32_MCi_STATUS = 0 AND OVER Flag in IA32_MCi_STATUS=0
                            THEN (* It is a spurious MCA Log. Log and clear the register *)
                                CLEAR_MC_BANK = TRUE;
                                GOTO LOG MCA REGISTER;
                        FI;
                        IF PCC = 1 and EN = 1 in IA32_MCi_STATUS
                            THEN (* processor context might have been corrupted *)
                                RESTARTABILITY = FALSE;
                            ELSE (* It is a uncorrected recoverable (UCR) error *)
                                IF S Flag in IA32_MCi_STATUS = 0
                                    THEN
                                        IF AR Flag in IA32_MCi_STATUS = 0
                                            THEN (* It is a uncorrected no action required (UCNA) error *)
                                                GOTO CONTINUE; (* let CMCI and CMC polling handler to process *)
                                            ELSE
                                                RESTARTABILITY = FALSE; (* S=0, AR=1 is illegal *)
                                        FI
                                FI;
                                IF RESTARTABILITY = FALSE
                                    THEN (* no need to take recovery action if RESTARTABILITY is already false *)
                                        NOERROR = FALSE;
                                        GOTO LOG MCA REGISTER;
                                FI;
                                (* S in IA32_MCi_STATUS = 1 *)
                                IF AR Flag in IA32_MCi_STATUS = 1
                                    THEN (* It is a software recoverable and action required (SRAR) error *)
                                        IF OVER Flag in IA32_MCi_STATUS = 1
```

```
                                          THEN
                                               RESTARTABILITY = FALSE;
                                               NOERROR = FALSE;
                                               GOTO LOG MCA REGISTER;
                                     FI
                                     IF MCACOD Value in IA32_MCi_STATUS is recognized
                                        AND Current Processor is an Affected Processor
                                          THEN
                                               Implement MCACOD specific recovery action;
                                               CLEAR_MC_BANK = TRUE;
                                          ELSE
                                               RESTARTABILITY = FALSE;
                                     FI;
                                ELSE (* It is a software recoverable and action optional (SRAO) error *)
                                     IF OVER Flag in IA32_MCi_STATUS = 0 AND
                                      MCACOD in IA32_MCi_STATUS is recognized
                                          THEN
                                               Implement MCACOD specific recovery action;
                                     FI;
                                     CLEAR_MC_BANK = TRUE;
                           FI; AR
                      FI; PCC
                      NOERROR = FALSE;
                      GOTO LOG MCA REGISTER;
                 ELSE  (* It is a corrected error; continue to the next IA32_MCi_STATUS *)
                      GOTO CONTINUE;
            FI; UC
       FI; VAL
LOG MCA REGISTER:
       SAVE IA32_MCi_STATUS;
       If MISCV in IA32_MCi_STATUS
            THEN
                 SAVE IA32_MCi_MISC;
       FI;
       IF ADDRV in IA32_MCi_STATUS
            THEN
                 SAVE IA32_MCi_ADDR;
       FI;
       IF CLEAR_MC_BANK = TRUE
            THEN
                 SET all 0 to IA32_MCi_STATUS;
                 If MISCV in IA32_MCi_STATUS
                      THEN
                           SET all 0 to IA32_MCi_MISC;
                 FI;
                 IF ADDRV in IA32_MCi_STATUS
                      THEN
                           SET all 0 to IA32_MCi_ADDR;
                 FI;
       FI;
       CONTINUE:
   OD;
( *END FOR *)
RETURN;
(* End of MCA ERROR PROCESSING*)
```

17.10.4.2 Corrected Machine-Check Handler for Error Recovery

When writing a corrected machine check handler, which is invoked as a result of CMCI or called from an OS CMC Polling dispatcher, consider the following:

- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank does not contain valid error information and does not need to be checked.

- The CMCI or CMC polling handler is responsible for logging and clearing corrected errors. The UC flag in each IA32_MCi_Status register indicates whether the reported error was corrected (UC=0) or not (UC=1).
- When IA32_MCG_CAP [24] is one, the CMC handler is also responsible for logging and clearing uncorrected no-action required (UCNA) errors. When the UC flag is one but the PCC, S, and AR flags are zero in the IA32_MCi_STATUS register, the reported error in this bank is an uncorrected no-action required (UCNA) error. In cases when SRAO error are signaled as UCNA error via CMCI, software can perform recovery for those errors identified in Table 17-16.
- In addition to corrected errors and UCNA errors, the CMC handler optionally logs uncorrected (UC=1 and PCC=1), software recoverable machine check errors (UC=1, PCC=0 and S=1), but should avoid clearing those errors from the MC banks. Clearing these errors may result in accidentally removing these errors before these errors are actually handled and processed by the MCE handler for attempted software error recovery.

Example 17-5 gives pseudocode for a CMCI handler with UCR support.

Example 17-5. Corrected Error Handler Pseudocode with UCR Support

```
Corrected Error HANDLER: (* Called from CMCI handler or OS CMC Polling Dispatcher*)
IF CPU supports MCA
    THEN
        FOR each bank of machine-check registers
            DO
                READ IA32_MCi_STATUS;
                IF VAL flag in IA32_MCi_STATUS = 1
                    THEN
                        IF UC Flag in IA32_MCi_STATUS = 0 (* It is a corrected error *)
                            THEN
                                GOTO LOG CMC ERROR;
                            ELSE
                                IF Bit 24 in IA32_MCG_CAP = 0
                                    THEN
                                        GOTO CONTINUE;
                                FI;
                                IF S Flag in IA32_MCi_STATUS = 0 AND AR Flag in IA32_MCi_STATUS = 0
                                    THEN (* It is a uncorrected no action required error *)
                                        GOTO LOG CMC ERROR
                                FI
                                IF EN Flag in IA32_MCi_STATUS = 0
                                    THEN (* It is a spurious MCA error *)
                                        GOTO LOG CMC ERROR
                                FI;
                        FI;
                FI;
                GOTO CONTINUE;
            LOG CMC ERROR:
                SAVE IA32_MCi_STATUS;
                If MISCV Flag in IA32_MCi_STATUS
                    THEN
                        SAVE IA32_MCi_MISC;
                        SET all 0 to IA32_MCi_MISC;
                FI;
                IF ADDRV Flag in IA32_MCi_STATUS
                    THEN
                        SAVE IA32_MCi_ADDR;
                        SET all 0 to IA32_MCi_ADDR
                FI;
                SET all 0 to IA32_MCi_STATUS;
                CONTINUE:
            OD;
    ( *END FOR *)
FI;
```

11. Updates to Chapter 21, Volume 3B

Change bars and violet text show changes to Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter:

- Moved Section 21.9.11, “Auto Counter Reload,” to Section 21.10.

CHAPTER 21
PERFORMANCE MONITORING

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

NOTE

Performance monitoring events can be found here: https://perfmon-events.intel.com/.

Additionally, performance monitoring event files for Intel processors are hosted by the Intel Open Source Technology Center. These files can be downloaded here: https://download.01.org/perfmon/.

21.1 PERFORMANCE MONITORING OVERVIEW

Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

The performance monitoring mechanisms and performance events defined for the Pentium, P6 family, and Intel processors based on Intel NetBurst microarchitecture are not architectural. They are all model specific (not compatible among processor families). Intel Core Solo and Intel Core Duo processors support a set of architectural performance events and a set of non-architectural performance events. Newer Intel processor generations support enhanced architectural performance events and non-architectural performance events.

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 21.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)." Non-architectural events for a given microarchitecture cannot be enumerated using CPUID; and they can be found at: https://perfmon-events.intel.com/ or at https://github.com/intel/perfmon/.

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and Interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 21.2.

See also:

- — Section 21.2, "Architectural Performance Monitoring."
- — Section 21.3, "Performance Monitoring (Intel® Core™ Processors and Intel® Xeon® Processors)."
  - Section 21.3.1, "Performance Monitoring for Processors Based on Nehalem Microarchitecture."
  - Section 21.3.2, "Performance Monitoring for Processors Based on Westmere Microarchitecture."
  - Section 21.3.3, "Intel® Xeon® Processor E7 Family Performance Monitoring Facility."
  - Section 21.3.4, "Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture."
  - Section 21.3.5, "3rd Generation Intel® Core™ Processor Performance Monitoring Facility."

    - Section 21.3.6, "4th Generation Intel® Core™ Processor Performance Monitoring Facility."
    - Section 21.3.7, "5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility."
    - Section 21.3.8, "6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility."
    - Section 21.3.9, "10th Generation Intel® Core™ Processor Performance Monitoring Facility."
    - Section 21.3.10, "12th and 13th Generation Intel® Core™ Processors, and 4th and 5th Generation Intel® Xeon® Scalable Processor Family Performance Monitoring Facility."
    - Section 21.3.11, "Intel® Series 2 Core™ Ultra Processor Performance Monitoring Facility."
- — Section 21.4, "Performance monitoring (Intel® Xeon™ Phi Processors)."
    - Section 21.4.1, "Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring."
- — Section 21.5, "Performance Monitoring (Intel Atom® Processors)."
    - Section 21.5.1, "Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)."
    - Section 21.5.2, "Performance Monitoring for Silvermont Microarchitecture."
    - Section 21.5.3, "Performance Monitoring for Goldmont Microarchitecture."
    - Section 21.5.4, "Performance Monitoring for Goldmont Plus Microarchitecture."
    - Section 21.5.5, "Performance Monitoring for Tremont Microarchitecture."
- — Section 21.6, "Performance Monitoring (Legacy Intel Processors)."
    - Section 21.6.1, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)."
    - Section 21.6.2, "Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)."
    - Section 21.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)."
    - Section 21.6.4, "Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture."
    - Section 21.6.4.5, "Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture."
    - Section 21.6.5, "Performance Monitoring and Dual-Core Technology."
    - Section 21.6.6, "Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache."
    - Section 21.6.7, "Performance Monitoring on L3 and Caching Bus Controller Sub-Systems."
    - Section 21.6.8, "Performance Monitoring (P6 Family Processor)."
    - Section 21.6.9, "Performance Monitoring (Pentium Processors)."
- — Section 21.7, "Counting Clocks."
- — Section 21.8, "IA32_PERF_CAPABILITIES MSR Enumeration."
- — Section 21.9, "PEBS Facility."

21.2 ARCHITECTURAL PERFORMANCE MONITORING

Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T

7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture. Intel Atom processors based on the Goldmont and Goldmont Plus microarchitectures support versionID 4. Intel Atom processors starting with processors based on the Tremont microarchitecture support versionID 5.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microarchitectures support version ID 3. Intel processors based on the Skylake through Coffee Lake microarchitectures support versionID 4. Intel processors starting with processors based on the Ice Lake microarchitecture support versionID 5.

21.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- The bit field layout of IA32_PERFEVTSELx is consistent across microarchitectures. A non-zero write of a field that is introduced after the initial implementation of architectural performance monitoring (Version 1) results in #GP if that field is not supported.
- Addresses of IA32_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32_PMC MSRs remain the same across microarchitectures.
- Each logical processor has its own set of IA32_PERFEVTSELx and IA32_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available to software in a logical processor (each IA32_PERFEVTSELx MSR is paired to the corresponding IA32_PMCx MSR).
- Number of bits supported in each IA32_PMCx.
- Number of architectural performance monitoring events supported in a logical processor.

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32_PERFEVTSELx/ IA32_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8]. Note that this may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

- IA32_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block. Note the number of IA32_PERFEVTSELx MSRs may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.
- The bit width of an IA32_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.
- Bit field layout of IA32_PERFEVTSELx MSRs is defined architecturally.

See Figure 21-1 for the bit field layout of IA32_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7) —** Selects the event logic unit used to detect microarchitectural conditions (see Table 21-3, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

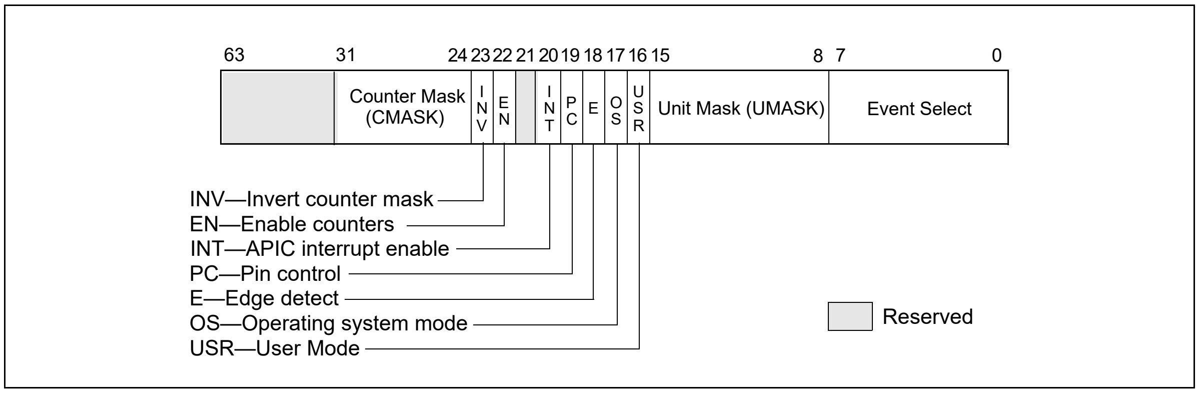


Figure 21-1. Layout of IA32_PERFEVTSELx MSRs

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

  A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 21-3; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX.
- **USR (user mode) flag (bit 16) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.
- **OS (operating system mode) flag (bit 17) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- **E (edge detect) flag (bit 18) —** Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

  This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
- **PC (pin control) flag (bit 19) —** Beginning with Sandy Bridge microarchitecture, this bit is reserved (not writeable). On processors based on previous microarchitectures, the logical processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.

- **INT (APIC interrupt enable) flag (bit 20) —** When set, the logical processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32_PERFEVTSELx[bit 22] = 0, before writing to IA32_PMCx.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31) —** When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

  This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

NOTE

A non-zero write of a field that is introduced in a later architectural performance monitoring version results in a general-protection (#GP) exception if that field is not supported by prior versions.

21.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32_FIXED_CTR0 through IA32_-FIXED_CTR2). Each of the fixed-function PMC can count only one architectural performance event.

  Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32_FIXED_CTR_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32_PMCx) via UMASK field in (IA32_PERFEVTSELx), configuring, programming IA32_FIXED_CTR_CTRL for fixed-function PMCs do not require any UMASK.
- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - IA32_PERF_GLOBAL_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or any general-purpose PMCs via a single WRMSR.
  - IA32_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - IA32_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - IA32_DEBUGCTL.Freeze_LBR_On_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 19.4.7 for details of the legacy Freeze LBRs on PMI control.
  - IA32_DEBUGCTL.Freeze_PerfMon_On_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 19.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

NOTE

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32_FIXED_CTR_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 21-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

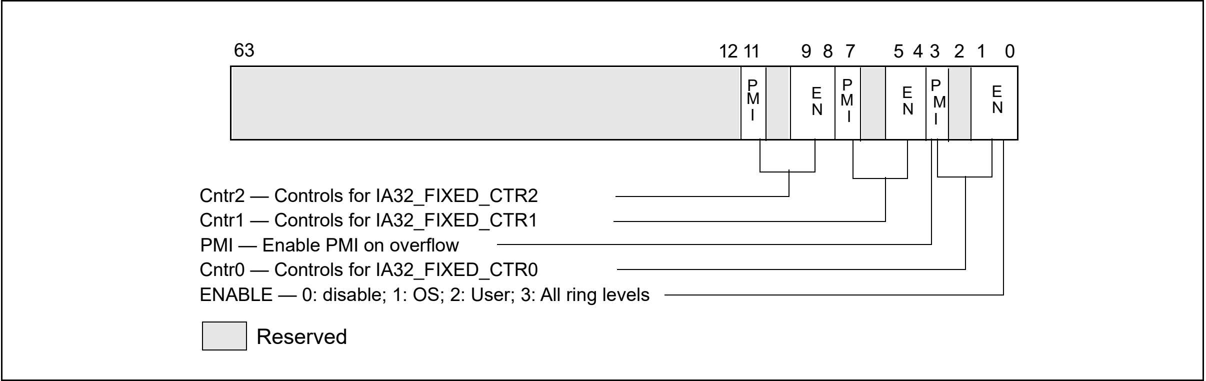

63
12 11
9 8 7
5 4 3 2 1 0
PMI
EN
PMI
EN
PMI
EN
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow
Cntr0 — Controls for IA32_FIXED_CTR0
ENABLE — 0: disable; 1: OS; 2: User; 3: All ring levels
Reserved

Figure 21-2. Layout of IA32_FIXED_CTR_CTRL MSR

- **Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- **PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 21-3 shows the layout of IA32_PERF_GLOBAL_CTRL. Each enable bit in IA32_PERF_GLOBAL_CTRL is ANDed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the ANDed results is true; counting is disabled when the result is false.

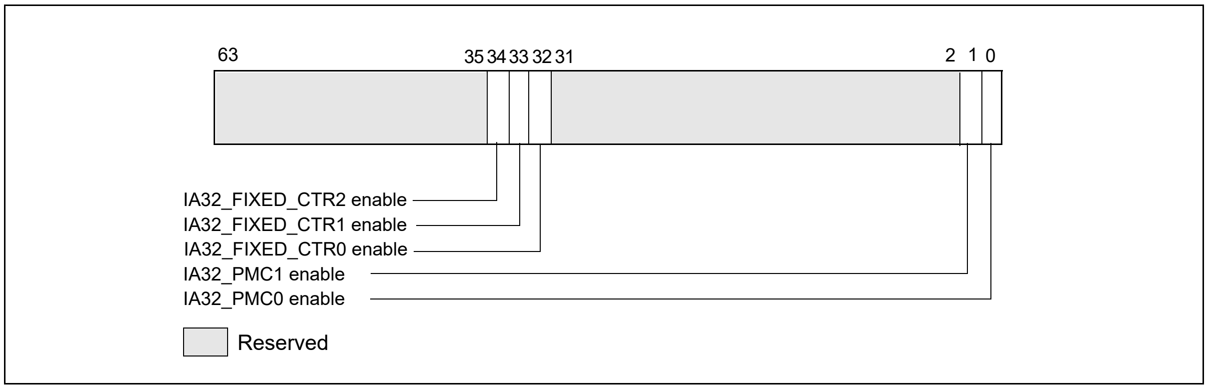


Figure 21-3. Layout of IA32_PERF_GLOBAL_CTRL MSR

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.

Table 21-1. Association of Fixed-Function Performance Counters with Architectural Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_FIXED_CTR0	309H	INST_RETIRED.ANY	This event counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The counter continues counting during hardware interrupts, traps, and in-side interrupt handlers.
IA32_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD CPU_CLK_UNHALTED.CORE	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.
IA32_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.
IA32_FIXED_CTR3	30CH	TOPDOWN.SLOTS	This event counts the number of available slots for an unhalted logical processor. The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core. Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

Table 21-1. Association of Fixed-Function Performance Counters with Architectural Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_FIXED_CTR4[1]	30DH	TOPDOWN_BAD_SPECULATION	This event counts Topdown slots that were not consumed by the backend due to a pipeline flush, such as a mispredicted branch or a machine clear. It provides a value equivalent to a general-purpose counter configured with UMask=00H and EventSelect=73H.
IA32_FIXED_CTR5[1]	30EH	TOPDOWN_FE_BOUND	This event counts Topdown slots where uops were not provided to the backend due to frontend limitations, such as instruction cache/TLB miss delays or decoder limitations. It provides a value equivalent to a general purpose counter configured with UMask=01H and EventSelect=9CH.
IA32_FIXED_CTR6[1]	30FH	TOPDOWN_RETIRING	This event counts Topdown slots that were committed (retired) by the backend. It provides a value equivalent to a general purpose counter configured with UMask=02H and EventSelect=C2H.

NOTES:

1. If this counter is supported, it will be accessible in the following MSRs: IA32_PERF_GLOBAL_STATUS (38EH), IA32_PERF_GLOBAL_CTRL (38FH), IA32_PERF_GLOBAL_STATUS_RESET (390H), and IA32_PERF_GLOBAL_STATUS_SET (391H).

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 21-4 shows the layout of IA32_PERF_GLOBAL_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status., and sets the “OvfBuffer” bit in IA32_PERF_-GLOBAL_STATUS.

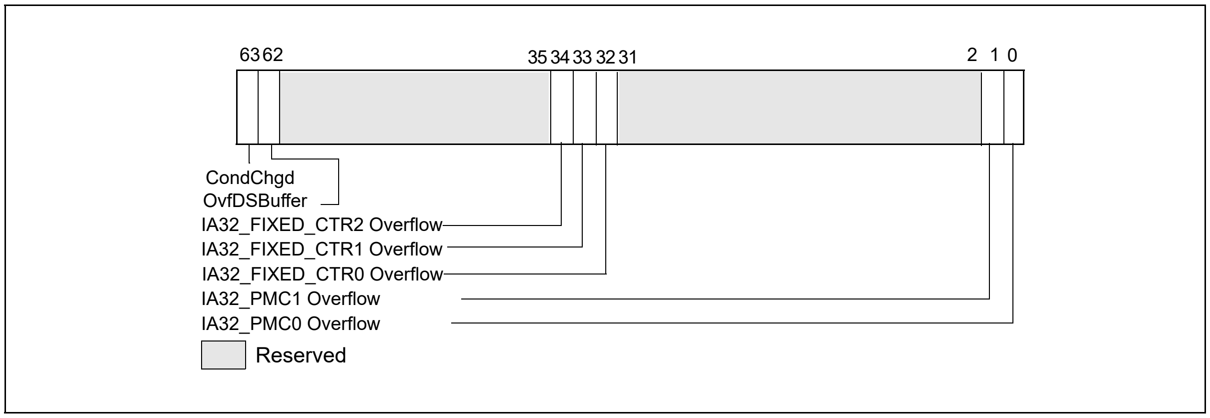


Figure 21-4. Layout of IA32_PERF_GLOBAL_STATUS MSR

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.

- Disabling event counting or interrupt-based event sampling.

The layout of IA32_PERF_GLOBAL_OVF_CTL is shown in Figure 21-5.

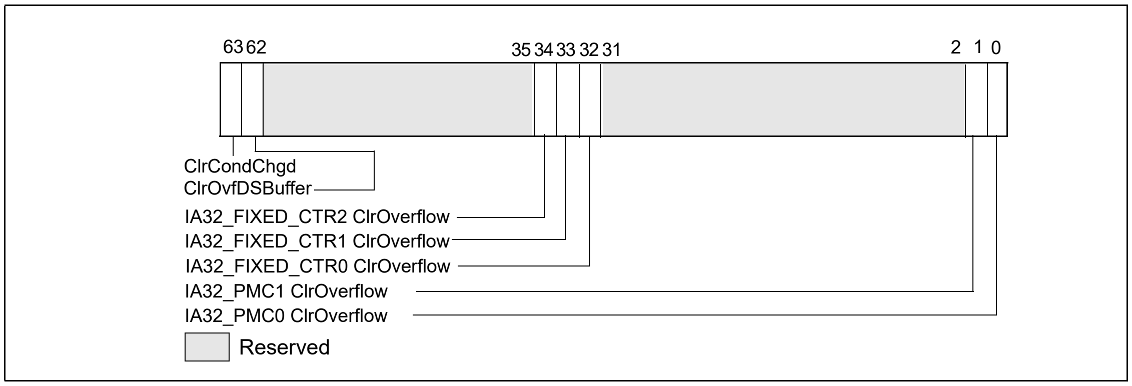


Figure 21-5. Layout of IA32_PERF_GLOBAL_OVF_CTRL MSR

21.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e., a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- AnyThread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  - — Each IA32_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 21-6.

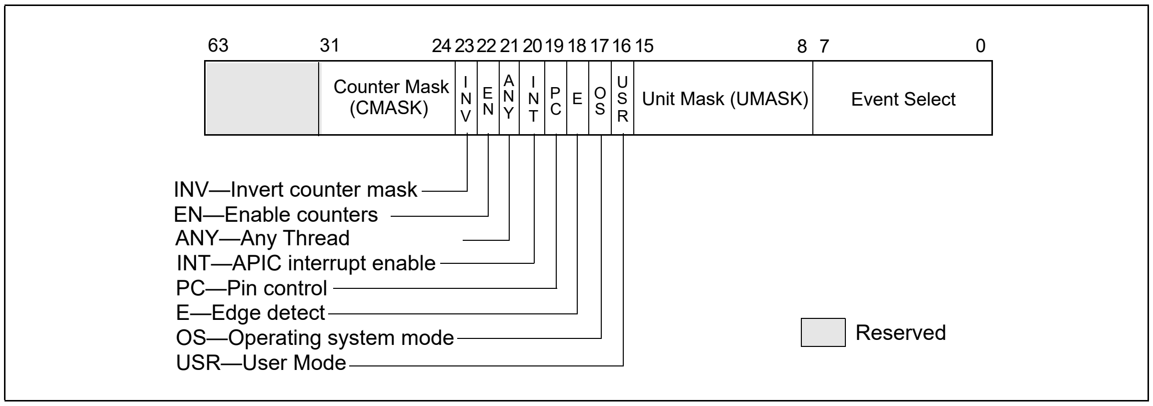


Figure 21-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

Bit 21 (AnyThread) of IA32_PERFEVTSELx is supported in architectural performance monitoring version 3 for processor core comprising of two or more logical processors. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring across all logical processors sharing a processor core. When bit 21 is 0, the counter only increments the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring in the logical processor which programmed the IA32_PERFEVTSELx MSR.

- — Each fixed-function performance counter IA32_FIXED_CTRx (starting at MSR address 309H) is configured by a 4-bit control block in the IA32_PERF_FIXED_CTR_CTRL MSR. The control block also allows thread-specificity configuration using an AnyThread bit for fixed-function counters 0, 1, and 2. The layout of IA32_PERF_FIXED_CTR_CTRL MSR is shown.

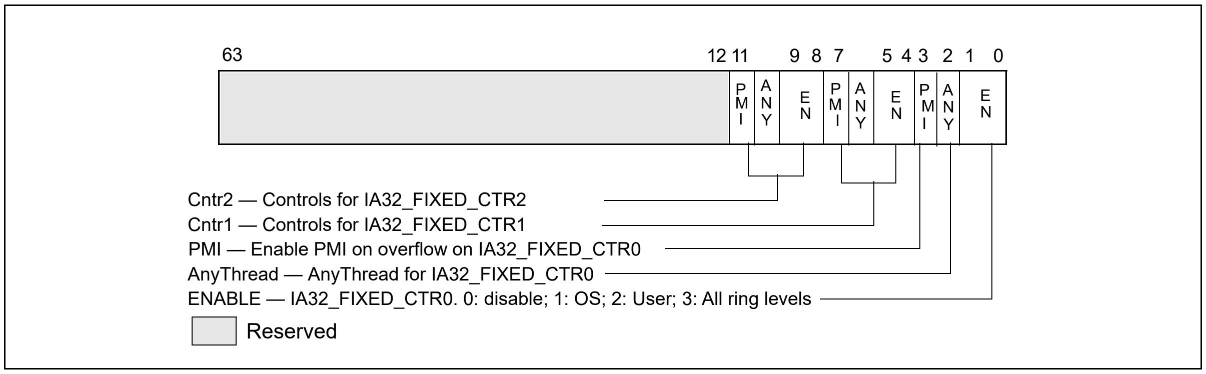

63
12 11
9 8 7
5 4 3 2 1 0
PMI
ANY
EN
PMI
ANY
EN
PMI
ANY
EN
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow on IA32_FIXED_CTR0
AnyThread — AnyThread for IA32_FIXED_CTR0
ENABLE — IA32_FIXED_CTR0. 0: disable; 1: OS; 2: User; 3: All ring levels
Reserved

Figure 21-7. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

Each control block for a fixed-function performance counter provides an **AnyThread** (bit position 2 + 4*N, N= 0, 1, etc.) bit. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the ENABLE setting of the corresponding control block of IA32_PERF_FIXED_CTR_CTRL) occurring across all logical processors sharing a processor core. When an **AnyThread** bit is 0 in IA32_PERF_FIXED_CTR_CTRL, the corresponding fixed-function counter only increments the associated event conditions occurring in the logical processor which programmed the IA32_PERF_FIXED_CTR_CTRL MSR.

- The IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, IA32_PERF_GLOBAL_OVF_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 21-8 and Figure 21-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.

NOTE

The number of general-purpose performance monitoring counters (i.e., N in Figure 21-9) can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology, (e.g., N=2 for 45 nm Intel Atom processors; N =4 for processors based on the Nehalem microarchitecture; for processors based on the Sandy Bridge microarchitecture, N = 4 if Intel Hyper Threading Technology is active and N=8 if not active). In addition, the number of counters may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

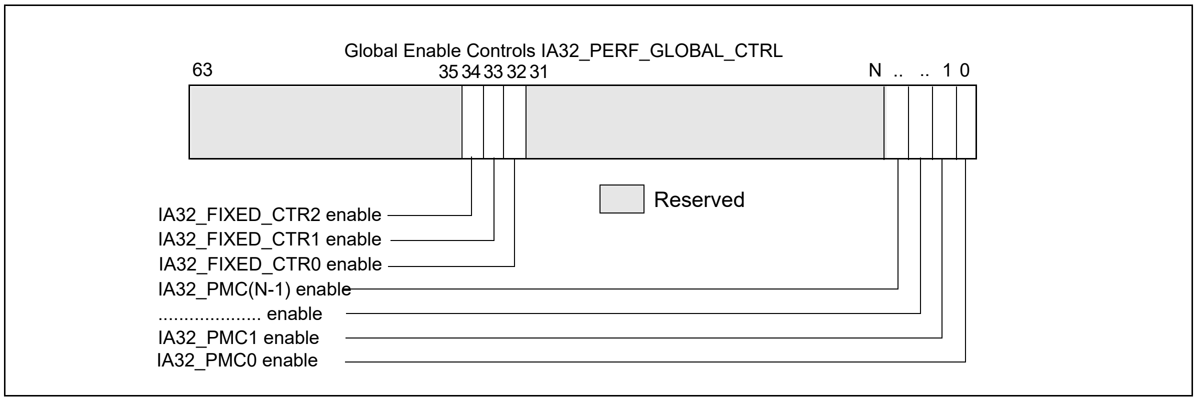


Figure 21-8. Layout of Global Performance Monitoring Control MSR

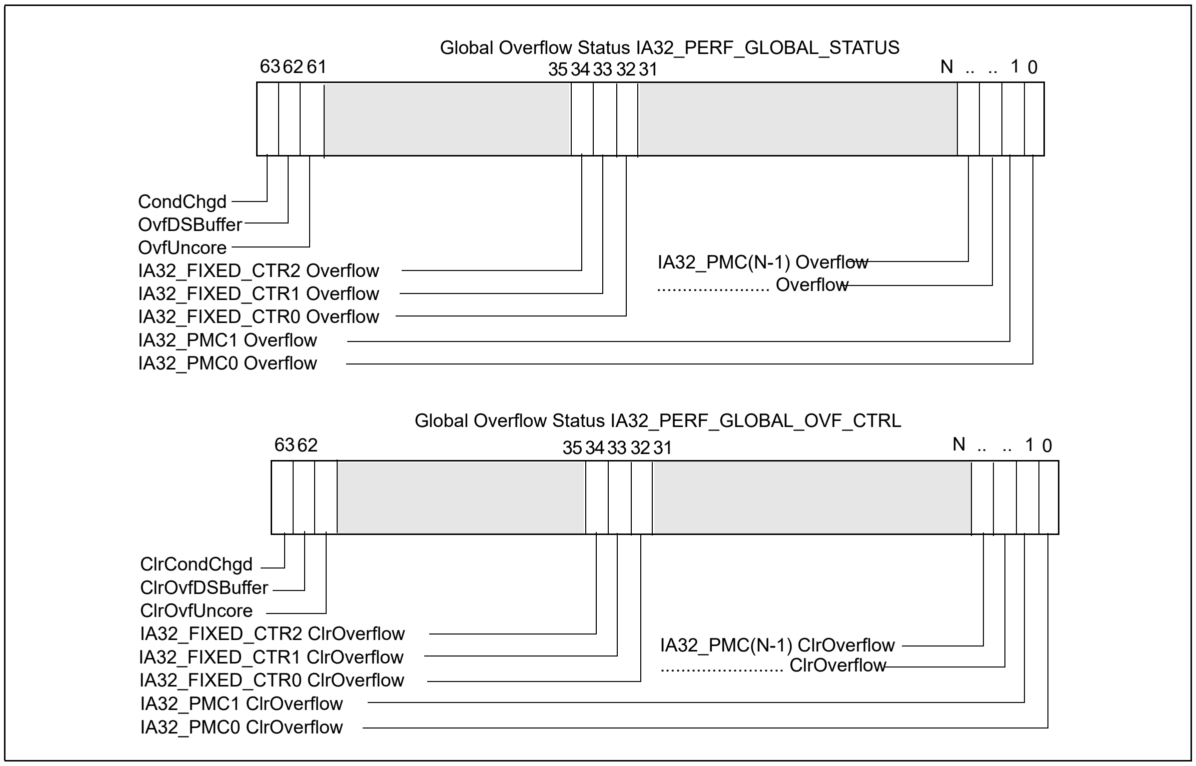


Figure 21-9. Global Performance Monitoring Overflow Status and Control MSRs

21.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL. While AnyThread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.

One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 25, “Introduction to Virtual Machine Extensions”) where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow an individual VM to employ performance monitoring facilities to profiles the performance characteristics of a workload. The use of the Anythread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- Configure the MSR bitmap to cause VM-exits for WRMSR to IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in VMX non-Root operation (see Chapter 26 for additional information),
- Clear the AnyThread bit of IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in the MSR-load lists for VM exits and VM entries (see Chapter 26, Chapter 28, and Chapter 29).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted.

21.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32_DEBUGCTL.Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI. Specifically version 4 provides the following enhancements:

- New indicators (LBR_FRZ, CTR_FRZ) in IA32_PERF_GLOBAL_STATUS, see Section 21.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32_DEBUGCTL.Freeze_LBRs_On_PMI and IA32_DEBUGCTL.Freeze_PerfMon_On_PMI: see Section 21.2.4.1. Legacy semantics of Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32_PERF_GLOBAL_STATUS and read-only performance counter enabling interface in IA32_PERF_GLOBAL_STATUS: see Section 21.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

21.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS

The IA32_PERF_GLOBAL_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32_PERF_GLOBAL_STATUS.LBR_FRZ[bit 58]: This bit is set due to the following conditions:
  - IA32_DEBUGCTL.FREEZE_LBR_ON_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.LBR_FRZ bit also serves as a control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural PerfMon version 4 or higher:
  - (IA32_DEBUGCTL.LBR & (!IA32_PERF_GLOBAL_STATUS.LBR_FRZ)) =1
- IA32_PERF_GLOBAL_STATUS.CTR_FRZ[bit 59]: This bit is set due to the following conditions:
  - IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.CTR_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural PerfMon version 4 or higher:
  - (IA32_PERFEVTSELn.EN & IA32_PERF_GLOBAL_CTRL.PMCn & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for programmable counter 'n', or
  - (IA32_PERF_FIXED_CRTL.ENi & IA32_PERF_GLOBAL_CTRL.FCi & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for fixed counter 'i'

The read-only enable interface IA32_PERF_GLOBAL_STATUS.CTR_FRZ provides a more efficient flow for a PMI handler to use IA32_DEBUGCTL.Freeze_PerfMon_On_PMI to filter out data that may distort target workload analysis, see Table 19-3. It should be noted the IA32_PERF_GLOBAL_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32_PERF_GLOBAL_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls, and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g., race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32_PERF_GLOBAL_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support it (for additional information about Intel SGX, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D):

- IA32_PERF_GLOBAL_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e., IA32_PMCx or IA32_FIXED_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32_PERF_GLOBAL_STATUS.ASCI was cleared).

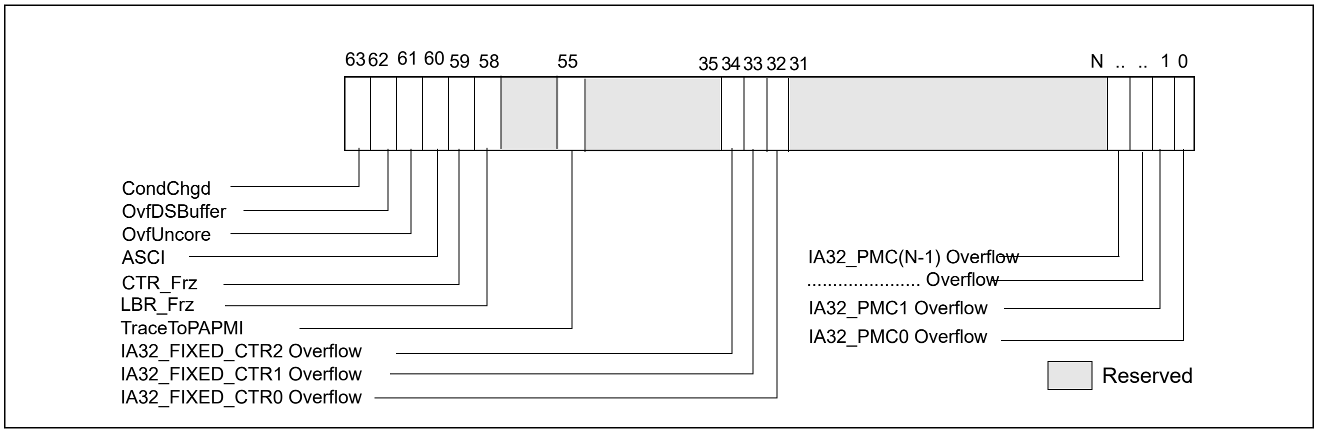


Figure 21-10. IA32_PERF_GLOBAL_STATUS MSR and Architectural PerfMon Version 4

Note, a processor's support for IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32_PERF_GLOBAL_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.

21.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32_PERF_-GLOBAL_STATUS MSR by software is done via IA32_PERF_GLOBAL_OVF_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32_PERF_-GLOBAL_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32_PERF_GLOBAL_OVF_CTRL is inherited by IA32_PERF_-GLOBAL_STATUS_RESET (see Figure 21-11). Further, IA32_PERF_GLOBAL_STATUS_RESET provides additional bit fields to clear the new indicators in IA32_PERF_GLOBAL_STATUS described in Section 21.2.4.1.

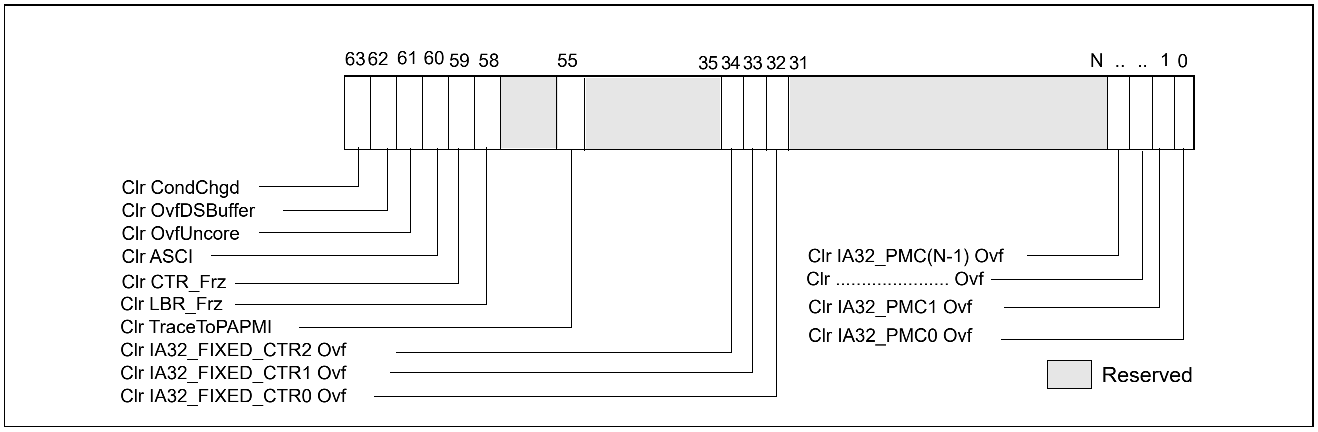


Figure 21-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural PerfMon Version 4

The IA32_PERF_GLOBAL_STATUS_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32_PERF_GLOBAL_STATUS. The IA32_PERF_GLOBAL_STATUS_SET interface can be used by a VMM to virtualize the state of IA32_PERF_GLOBAL_STATUS across VMs.

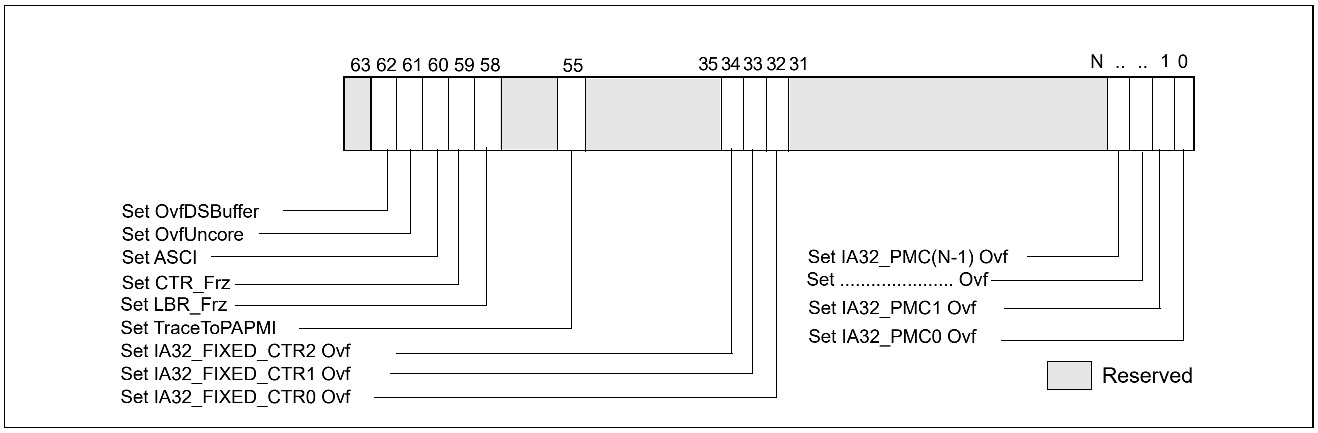


Figure 21-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural PerfMon Version 4

21.2.4.3 IA32_PERF_GLOBAL_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor's performance monitoring facilities. The IA32_MISC_ENABLE.PERFMON_AVAILABLE[bit 7] interface could not serve the need of multiple agent adequately. A white paper, "Performance Monitoring Unit Sharing Guideline"[1], proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32_PERF_GLOBAL_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32_PERF_GLOBAL_INUSE is shown in Figure 21-13.

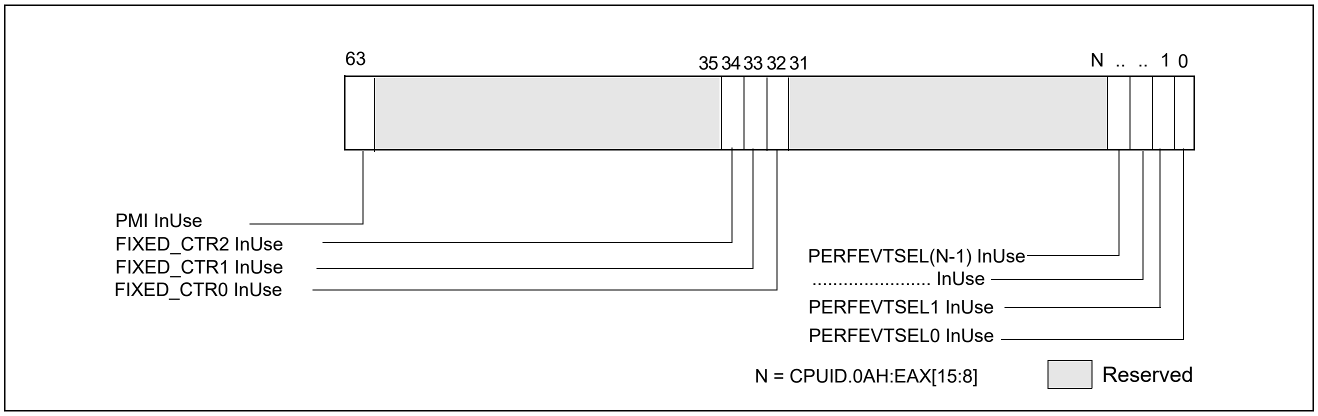


Figure 21-13. IA32_PERF_GLOBAL_INUSE MSR and Architectural PerfMon Version 4

The IA32_PERF_GLOBAL_INUSE MSR provides an "InUse" bit for each programmable performance counter and fixed counter in the processor. Additionally, it includes an indicator if the PMI mechanism has been configured by a profiling agent.

- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL0_InUse[bit 0]: This bit reflects the logical state of (IA32_PERFEVTSEL0[7:0] != 0).

1. Available at http://www.intel.com/sdm

- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL1_InUse[bit 1]: This bit reflects the logical state of (IA32_PERFEVTSEL1[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL2_InUse[bit 2]: This bit reflects the logical state of (IA32_PERFEVTSEL2[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSELn_InUse[bit n]: This bit reflects the logical state of (IA32_PERFEVTSELn[7:0] != 0), n < CPUID.0AH:EAX[15:8].
- IA32_PERF_GLOBAL_INUSE.FC0_InUse[bit 32]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[1:0] != 0).
- IA32_PERF_GLOBAL_INUSE.FC1_InUse[bit 33]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[5:4] != 0).
- IA32_PERF_GLOBAL_INUSE.FC2_InUse[bit 34]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[9:8] != 0).
- IA32_PERF_GLOBAL_INUSE.PMI_InUse[bit 63]: This bit is set if any one of the following bit is set:
  — IA32_PERFEVTSELn.INT[bit 20], n < CPUID.0AH:EAX[15:8].
  — IA32_FIXED_CTR_CTRL.ENi_PMI, i = 0, 1, 2.
  — Any IA32_PEBS_ENABLES bit which enables PEBS for a general-purpose or fixed-function performance counter.

21.2.5 Architectural Performance Monitoring Version 5

Processors supporting architectural performance monitoring version 5 also support versions 1, 2, 3, and 4, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 5 provides the following enhancements:

- Deprecation of AnyThread mode, see Section 21.2.5.1.
- Individual enumeration of Fixed counters in CPUID.0AH, see Section 21.2.5.2.
- Domain separation, see Section 21.2.5.3.

21.2.5.1 AnyThread Mode Deprecation

With Architectural Performance Monitoring Version 5, a processor that supports AnyThread mode deprecation is enumerated by CPUID.0AH.EDX[15]. If set, software will not have to follow guidelines in Section 21.2.3.1.

21.2.5.2 Fixed Counter Enumeration

With Architectural Performance Monitoring Version 5, register CPUID.0AH.ECX indicates Fixed Counter enumeration. It is a bit mask which enumerates the supported Fixed Counters in a processor. If bit 'i' is set, it implies that Fixed Counter 'i' is supported. Software is recommended to use the following logic to check if a Fixed Counter is supported on a given processor:

```
FxCtr[i]_is_supported := ECX[i] || (EDX[4:0] > i);
```

21.2.5.3 Domain Separation

When the INV flag in IA32_PERFEVTSELx is used, a counter stops counting when the logical processor exits the C0 ACPI C-state.

21.2.6 Architectural Performance Monitoring Version 6

Processors supporting architectural performance monitoring version 6 also support versions 1, 2, 3, 4, and 5, as well as the capabilities enumerated by CPUID leaves 0AH and 23H. Specifically, version 6 provides the following enhancements:

- PerfMon MSRs Aliasing, see Section 21.2.6.1.

- UnitMask2, see Section 21.2.6.2.
- Equal flag, see Section 21.2.6.3.

21.2.6.1 Performance Monitoring MSR Aliasing

Architectural performance monitoring version 6 includes a new range for the counters' MSRs in the 19xxH address range[1]. The new MSR range allows for scaling the number of general-purpose and fixed-function counters beyond the quantities in current products. Additionally, it banks registers of the same counter closer to each other.

All legacy and new counters, i.e., those enumerated in CPUID.(EAX = 23H, ECX = 01H), will be supported in this new address range. Moving forward, newer counters may be supported in the new address range, but not in the legacy one.

Table 21-2. New Performance Monitoring MSR Naming Details

Register	General Counter n	Fixed Counter m
Counter	IA32_PMC_GPn_CTR	IA32_PMC_FXm_CTR
Event-Select	IA32_PMC_GPn_CFG_A	N/A
Reload Config	IA32_PMC_GPn_CFG_B	IA32_PMC_FXm_CFG_B
Event-Select Extended	IA32_PMC_GPn_CFG_C	IA32_PMC_FXm_CFG_C

An IA32_PMC_GPn_CTR MSR can be used to access the counter value for a GP (general-purpose) counter 'n.' On processors that support CPUID leaf 23H, a general-purpose (GP) counter 'n' that is enumerated in both CPUID leaf 23H and leaf 0AH can be accessed through either IA32_PMC_GPn_CTR or the legacy MSR addresses (IA32_PMCn, IA32_A_PMCn). In contrast, a counter 'n' that is only enumerated in CPUID leaf 23H can only be accessed through IA32_PMC_GPn_CTR. This guideline also applies to the other MSR aliases described in this section (i.e., IA32_PMC_GPn_CFG_A and IA32_PERFEVTSELn, IA32_PMC_FXm_CTR and IA32_FIXED_CTRm). The IA32_PMC_GPn_CTR MSR address[2] for counter 'n' is 1900H + 4 * n, and this MSR has full-width write support.

The IA32_PMC_GPn_CFG_A MSR can be used to access the performance event select register for a GP counter 'n' and is at address[3] 1901H + 4 * n. The reload configuration MSRs for GP counter 'n,' IA32_PMC_GPn_CFG_B, is at MSR address 1902H + 4 * n. There is no legacy MSR alias to this reload configuration register. Thus, the register only exists when enumerated in CPUID leaf 23H. Similarly, no legacy MSR alias exists for the event-select extended registers, IA32_PMC_GPn_CFG_C, which are at MSR address 1903H + 4 * n for GP counter 'n.'

An IA32_PMC_FXm_CTR MSR can be used to access the counter value for a fixed-function counter 'm' if that counter is enumerated in CPUID leaf 23H. The IA32_PMC_FXm_CTR MSR address for fixed-function counter 'm' is 1980H + 4 * m. There is no alias for the fixed-function counters' reload configuration or event select extended registers (IA32_PMC_FXm_CFG_B at MSR addresses 1982H + 4 * m and IA32_PMC_FXm_CFG_C at MSR address 1983H + 4 * m, respectively).

The available general-purpose and fixed-function counters are reported by CPUID.(EAX = 23H, ECX = 01H):EAX and CPUID.(EAX = 23H, ECX = 01H):EBX, respectively.[4] Note that not all counters enumerated in CPUID leaf 23H may have corresponding IA32_PMC_GPn_CFG_B, IA32_PMC_GPn_CFG_C, IA32_PMC_FXm_CFG_B, or IA32_PMC_FXm_CFG_C MSRs. The enumeration and usage of these MSRs are described in Section 21.10, "Auto Counter Reload." The enumeration in CPUID leaf 23H is true-view, and thus, the enumeration may only be set on (and the MSRs/counters they enumerate only supported on) a subset of the logical processors of the system.

The architectural performance monitoring version 6 enhanced layout of the IA32_PERFEVTSELx MSR is shown in Figure 21-14.

1. This feature is also available in a subset of processors with a CPUID signature value of DisplayFamily_DisplayModel 06_C5H or 06_C6H (though they report IA32_PERF_CAPABILITIES.PEBS_FMT as 5).
2. As an example, the IA32_PMC_GP1_CTR MSR has MSR address 1904H. Note that the legacy full-width MSR addresses for the counters, IA32_A_PMCn MSRs, remains at MSR address 4C1H + n.
3. As an example, the IA32_PMC_GP1_CFG_A MSR has MSR address 1905H. Note that the legacy MSR address for the event select registers, IA32_PERFEVTSELn MSRs, remain at MSR address 186H + n.
4. The valid range of fixed-function counters is 0 through 15.

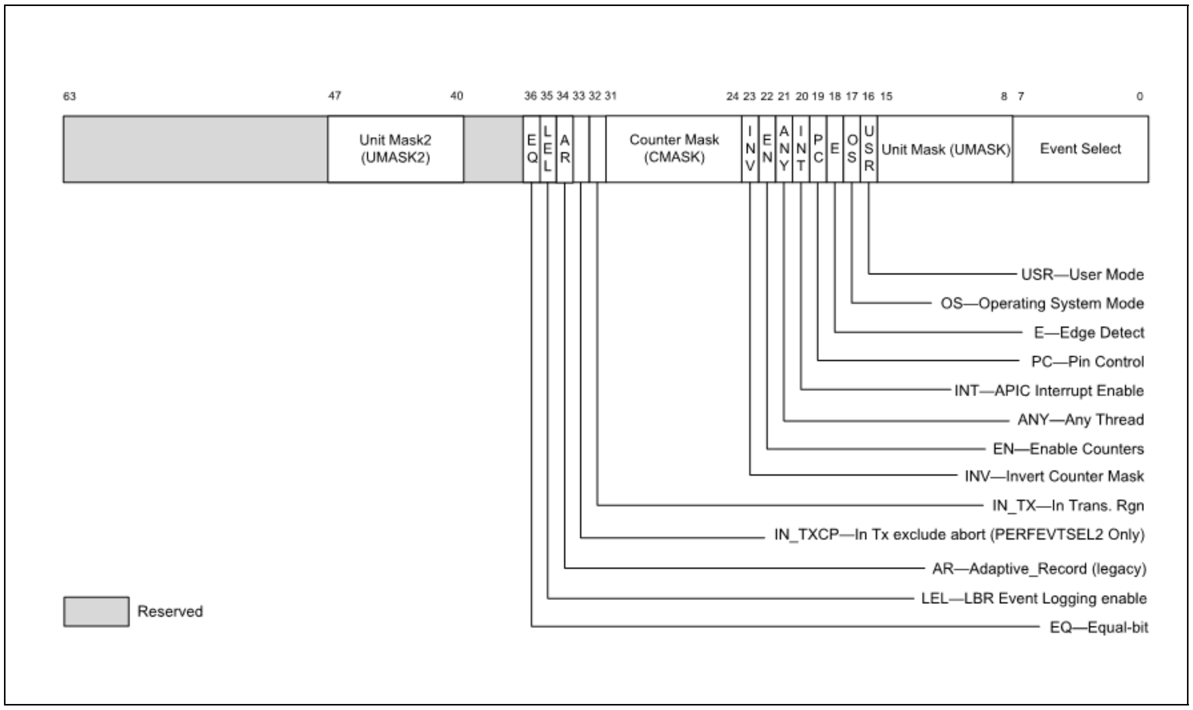


Figure 21-14. IA32_PMC_GPx_CFG_A MSR (also known as IA32_PERFEVTSELx) Supporting Architectural Performance Monitoring Version 6

NOTES

The EQ bit and UMASK2 field are added in Architectural Performance Monitoring Version 6.

The IN_TX and IN_TXCP bits are available only on processors supporting Intel TSX.

The architectural performance monitoring version 6 enhanced layout of the IA32_FIXED_CTR_CTRL MSR is shown in Figure 21-15.

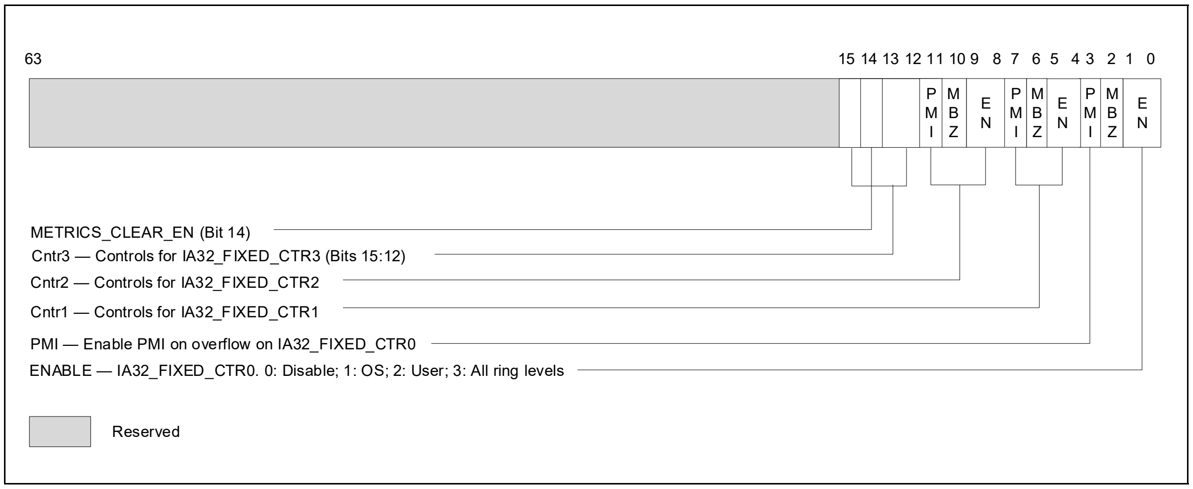

63
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
PMI
MBZ
EN
PMI
MBZ
EN
PMI
MBZ
EN
METRICS_CLEAR_EN (Bit 14)
Cntr3 — Controls for IA32_FIXED_CTR3 (Bits 15:12)
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow on IA32_FIXED_CTR0
ENABLE — IA32_FIXED_CTR0. 0: Disable; 1: OS; 2: User; 3: All ring levels
Reserved

Figure 21-15. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 6

21.2.6.2 Unit Mask 2

Architectural performance monitoring version 6 introduces a new Unit Mask 2 (UMASK2) field in the IA32_PERFEVTSELx MSRs. It is supported if enumerated by CPUID.(EAX=23H, ECX=0H):EBX[bit 0].

- UMASK2 field (bits 40 through 47): These bits qualify the condition that the selected event logic unit detects. Valid UMASK2 values for each event logic unit are specific to the unit. The new UMASK2 field may also be used in conjunction with UMASK.

21.2.6.3 Equal Flag

Architectural performance monitoring version 6 introduces a new Equal (EQ) flag in the IA32_PERFEVTSELx MSRs. It is supported if enumerated by CPUID.(EAX=23H, ECX=0H):EBX[bit 1].

- EQ flag (bit 36): When the EQ flag is set and the INV flag is clear, the comparison evaluates to true if the selected performance monitoring event (the event) is equal to the specified Counter Mask value (CMask). When the EQ flag is set and the INV flag is set, the comparison evaluates to true if the event is less than the CMask value and the event is not zero. Note that if the CMask is zero, the EQ flag is ignored.

21.2.7 Pre-defined Architectural Performance Events

Table 21-3 lists architecturally defined events.

Table 21-3. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

Bit Position CPUID.AH.EBX and CPUID.23H.03H.EAX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles[1]	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H
7	Topdown Slots	01H	A4H
8	Topdown Backend Bound	02H	A4H
9	Topdown Bad Speculation	00H	73H
10	Topdown Frontend Bound	01H	9CH
11	Topdown Retiring	02H	C2H
12	LBR Inserts	01H	E4H

NOTES:

1. Implementations prior to the 12th generation Intel® Core™ processor P-cores count at core crystal clock, TSC, or bus clock frequency.

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 21-3). The number of architectural events is reported through CPUID.0AH:EAX[31:24], while non-zero bits in CPUID.0AH:EBX indicate any architectural events that are not available.

The behavior of each architectural performance event is expected to be consistent on all processors that support that event. Minor variations between microarchitectures are noted below:

- **UnHalted Core Cycles —** Event select 3CH, Umask 00H

  This event counts core clock cycles when the clock signal on a specific core is running (not halted). The counter does not advance in the following conditions:

  - An ACPI C-state other than C0 for normal operation.
  - HLT.
  - STPCLK# pin asserted.
  - Being throttled by TM1.
  - During the frequency switching phase of a performance state transition (see Chapter 16, "Power and Thermal Management").

  The performance counter for this event counts across performance state transitions using different core clock frequencies.

- **Instructions Retired —** Event select C0H, Umask 00H

  This event counts the number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. An instruction with a REP prefix counts as one instruction (not per iteration). Faults before the retirement of the last micro-op of a multi-ops instruction are not counted.

  This event does not increment under VM-exit conditions. Counters continue counting during hardware interrupts, traps, and inside interrupt handlers.

- **UnHalted Reference Cycles —** Event select 3CH, Umask 01H

  This event counts reference clock cycles at a fixed frequency while the clock signal on the core is running. The event counts at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. Current implementations use the core crystal clock, TSC or the bus clock. Because the rate may differ between implementations, software should calibrate it to a time source with known frequency.

- **Last Level Cache References —** Event select 2EH, Umask 4FH

  This event counts requests originating from the core that reference a cache line in the last level on-die cache. The event count includes speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Last Level Cache Misses —** Event select 2EH, Umask 41H

  This event counts each cache miss condition for references to the last level on-die cache. The event count may include speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Branch Instructions Retired —** Event select C4H, Umask 00H

  This event counts branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction.

- **All Branch Mispredict Retired —** Event select C5H, Umask 00H

  This event counts mispredicted branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction in the branch prediction hardware.

  Branch prediction hardware is implementation-specific across microarchitectures; value comparison to estimate performance differences is not recommended.

- **Topdown Slots —** Event select A4H, Umask 01H

  This event counts the total number of available slots for an unhalted logical processor.

  The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core, in processors that support Intel Hyper-Threading Technology.

  Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

NOTE

Programming decisions or software precisians on functionality should not be based on the event values or dependent on the existence of performance monitoring events.

- **Topdown Backend Bound —** Event select A4H, Umask 02H

  This event counts a subset of the Topdown Slots event that was not consumed by the backend pipeline due to lack of backend resources, as a result of memory subsystem delays, execution unit limitations, or other conditions.

  The count may be distributed among unhalted logical processors that share the same physical core, in processors that support Intel® Hyper-Threading Technology.

  Software can use this event as the numerator for the Backend Bound metric (or top-level category) of the Topdown Microarchitecture Analysis method.

  Software can also derive the Backend Bound Slots using the formula: Backend Bound Slots = (Total Slots - Bad Speculation Slots - Frontend Bound Slots - Retiring Slots).
- **Topdown Bad Speculation —** Event select 73H, Umask 00H

  This event counts a subset of the Topdown Slots event that was wasted due to incorrect speculation as a result of incorrect control-flow or data speculation. Common examples include branch mispredictions and memory ordering clears.

  The count may be distributed among impacted logical processors that share the same physical core, for some processors that support Intel Hyper-Threading Technology.

  Software can use this event as the numerator for the Bad Speculation metric (or top-level category) of the Topdown Microarchitecture Analysis method.

  Software can also derive the Bad Speculation Slots using the formula: Bad Speculation Slots = (Total Slots - Backend Bound Slots - Frontend Bound Slots - Retiring Slots).
- **Topdown Frontend Bound —** Event select 9CH, Umask 01H

  This event counts a subset of the Topdown Slots event that had no operation delivered to the backend pipeline due to instruction fetch limitations when the backend could have accepted more operations. Common examples include instruction cache misses and x86 instruction decode limitations.

  The count may be distributed among unhalted logical processors that share the same physical core, in processors that support Intel Hyper-Threading Technology.

  Software can use this event as the numerator for the Frontend Bound metric (or top-level category) of the Topdown Microarchitecture Analysis method.
- **Topdown Retiring —** Event select C2H, Umask 02H

  This event counts a subset of the Topdown Slots event that is utilized by operations that eventually get retired (committed) by the processor pipeline. Usually, this event positively correlates with higher performance as measured by the instructions-per-cycle metric.

  Software can use this event as the numerator for the Retiring metric (or top-level category) of the Topdown Microarchitecture Analysis method.
- **LBR Inserts —** Event select E4H, Umask 01H

  This event counts when an LBR (Last Branch Record) entry is inserted or removed. Inserted means an actual LBR buffer update has occurred, considering LBR configuration and filtering. An LBR entry is removed when a RET instruction is retired in LBR Call-stack mode.

  Software may use this event in usages like profile-guided optimization (PGO) for profiling collections across Intel processors and in virtualized environments.

21.2.8 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32_PMCx are writable via WRMSR instruction. However, the value written into IA32_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32_PERF_CAPABILITIES[13] to enumerate its full-width-write capability See Figure 21-67.

If IA32_PERF_CAPABILITIES.FW_WRITE[bit 13] =1, each IA32_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32_A_PMC0.

The bit width of the performance monitoring counters is specified in CPUID.0AH:EAX[23:16].

If IA32_A_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32_A_PMCi will cause IA32_PMCi to be updated by:

```
COUNTERWIDTH = CPUID.0AH:EAX[23:16] bit width of the performance monitoring counter
IA32_PMCi[COUNTERWIDTH-1:32] := EDX[COUNTERWIDTH-33:0]);
IA32_PMCi[31:0] := EAX[31:0];
EDX[63:COUNTERWIDTH] are reserved
```

21.2.9 Scalable Enumeration Architecture

An Architectural Performance Monitoring Extended (ArchPerfMonExt) leaf 23H is added to the CPUID instruction for enhanced enumeration of PMU architectural features. Additionally, the IA32_PERF_CAPABILITIES MSR enhances enumeration for PMU non-architectural features.

NOTE

CPUID leaf 0AH continues to report useful attributes, such as architectural performance monitoring version ID and counter width (# bits).

CPUID leaf 23H enhances previous enumeration of PMU capabilities:

- Employs CPUID sub-leafing to accommodate future PMU extensions.
- Exposes true-view resources per logical processor.
- Introduces a bitmap (true-view) enumeration of general-purpose counters availability.
- A bitmap (true-view) enumeration of fixed-function counters availability.
- A bitmap (true-view) enumeration of architectural performance monitoring events.

Processors that support this enhancement set CPUID.(EAX=07H, ECX=01H):EAX.ArchPerfMonExt[bit 8].

21.2.9.1 CPUID Sub-Leafing

CPUID leaf 23H contains additional architectural PMU capabilities. This leaf supports sub-leafing, providing each distinct PMU feature with an individual sub-leaf for enumerating its details.

The availability of sub-leaves is enumerated via CPUID.(EAX=23H, ECX=0H):EAX. For each bit *n* set in this field, sub-leaf *n* under CPUID leaf 23H is supported.

21.2.9.2 Reporting Per Logical Processor

CPUID leaf 23H provides a true-view of per logical processor PMU capabilities. This leaf reports the actual support of the individual logical processor that the CPUID instruction was executed on; this applies to all sub-leaves.

Software must not make assumptions that CPUID leaf 23H would report any value the same on another logical processor. It is required to read CPUID leaf 23H on every logical processor and program that logical processor only according to the values returned by the CPUID leaf 23H directly executed upon it. It is a requirement of software to compare and determine common features between logical processors if required by iterating over each logical processor’s CPUID leaf 23H.

Conversely, CPUID leaf 0AH provides a maximum common set of capabilities across logical processors when a feature is not supported by all logical processors.

NOTE

Locating a PMU feature under CPUID leaf 23H alerts software that the feature may not be supported uniformly across all logical processors.

21.2.9.3 General-Purpose Counters Bitmap

CPUID.(EAX=23H, ECX=01H):EAX reports a bitmap for available general-purpose counters. (CPUID leaf 0AH reports only the total number of general-purpose counters.)

This capability enables a virtual-machine monitor to reserve lower-index counters for its own use, while exposing higher-index counters to guest software. This is especially important should the general-purpose counters not be fully homogeneous.

Software should utilize the new bitmap reporting, including for detecting the number of available general-purpose counters. To facilitate this transition, the number of general-purpose counters in CPUID leaf 0AH will not go beyond eight, even if the processor has support for more than eight general-purpose counters.

Note that general-purpose counters that are exclusively enumerated in CPUID.(EAX=23H, ECX=01H):EAX may not support the legacy MSR address range; see Section 21.2.6.1, "Performance Monitoring MSR Aliasing," for details.

21.2.9.4 Fixed-Function Counters True-View Bitmap

CPUID.(EAX=23H, ECX=01H):EBX reports a bitmap for available fixed-function counters. (CPUID leaf 0AH reports the common number of contiguous fixed-function counters in addition to a common bitmap of fixed-function counters availability.)[1]

This capability enables privileged software to expose per logical processor enumeration of fixed-function counters. This is especially important should the fixed-function counters not be available on all logical processors.

Note that programmable counters that are exclusively enumerated in CPUID.(EAX=23H, ECX=01H):EAX may not support the legacy MSR address range; see Section 21.2.6.1, "Performance Monitoring MSR Aliasing," for details.

21.2.9.5 Architectural Performance Monitoring Events Bitmap

CPUID.(EAX=23H, ECX=03H):EAX provides a true-view of per logical processor available architectural performance monitoring events. For each bit *n* set in this field, the processor supports Architectural Performance Monitoring Event of index *n* (positive polarity).

Conversely, CPUID.0AH:EBX provides a maximum common set of architectural performance monitoring events supported by all logical processors, where if bit *n* is set, it denotes the processor does not necessarily support Architectural Performance Monitoring Event of index *n* on all logical processors (negative polarity).

21.2.9.6 TMA Slots Per Cycle

CPUID.(EAX=23H, ECX=0H):ECX[7:0] reports the number of TMA slots per cycle in a true-view per logical-processor fashion.

This number can be multiplied by the number of cycles (from CPU_CLK_UNHALTED.THREAD / CPU_CLK_UNHALTED.CORE or IA32_FIXED_CTR1) to determine the total number of TMA slots.

Because of microarchitectural reasons, some logical processors may be reporting TMA slots per cycle as 0. In such situations, software can use other methods, like programmable events or fixed counters, to understand the performance issues.

1. The valid range of fixed-function counters is 0 through 15.

21.3 PERFORMANCE MONITORING (INTEL® CORE™ PROCESSORS AND INTEL® XEON® PROCESSORS)

21.3.1 Performance Monitoring for Processors Based on Nehalem Microarchitecture

Intel Core i7 processor family[1] supports architectural performance monitoring capability with version ID 3 (see Section 21.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Nehalem microarchitecture, and provides four general-purpose performance counters (IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_-FIXED_CTR1, IA32_FIXED_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as "uncore".
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 21-6 and described in Section and Section 21.2.3.

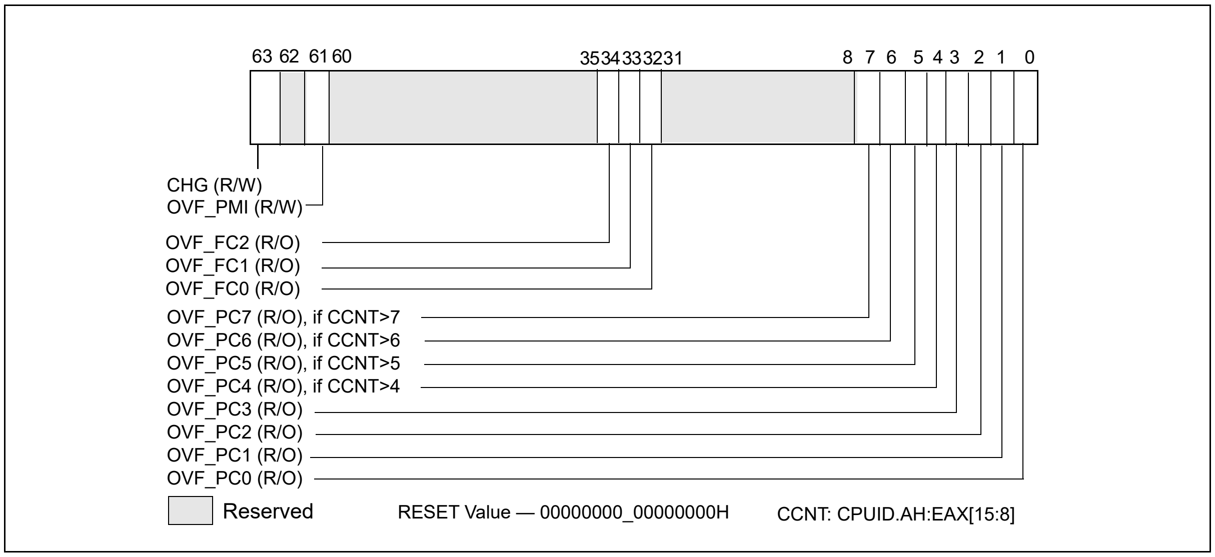


Figure 21-16. IA32_PERF_GLOBAL_STATUS MSR

1. Intel Xeon processor 5500 series and 3400 series are also based on Nehalem microarchitecture; the performance monitoring facilities described in this section generally also apply.

21.3.1.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32_PMCx, associated counter configuration MSRs, IA32_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Nehalem microarchitecture has been enhanced to include new data format to capture additional information, such as load latency.
- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Nehalem microarchitecture. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.

NOTE

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 21.2.1.

21.3.1.1.1 Processor Event Based Sampling (PEBS)

All general-purpose performance counters, IA32_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32_MISC_ENABLE[7] and IA32_MISC_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32_PEBS_ENABLE provides 4 bits that software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32_PEBS_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEBS_ENABLE for processors based on Nehalem microarchitecture is shown in Figure 21-17.

When a counter is enabled to capture machine state (PEBS_EN_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

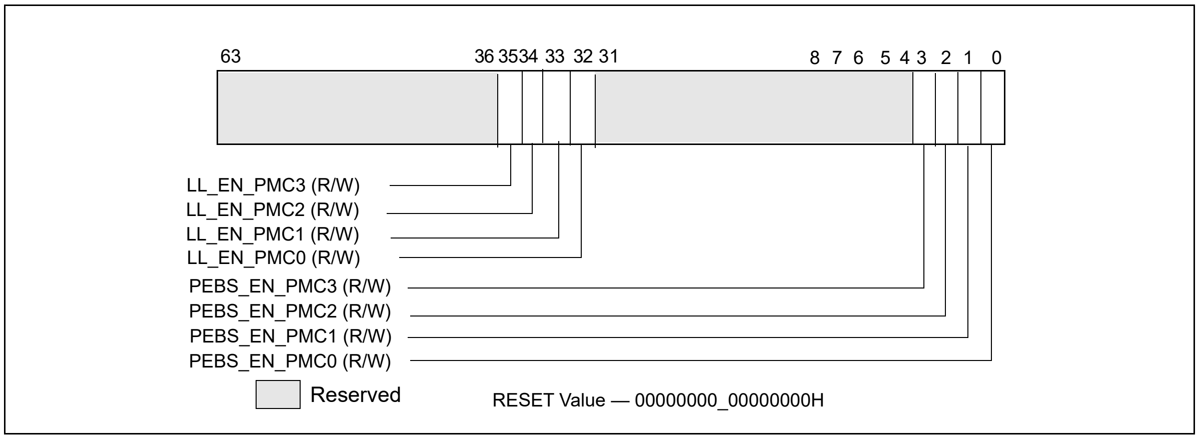


Figure 21-17. Layout of IA32_PEBS_ENABLE MSR

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32_PERF_CAPABILITIES[11:8] (see Figure 21-67).

The behavior of PEBS assists is reported by IA32_PERF_CAPABILITIES[6] (see Figure 21-67). The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

mov eax, [eax] ; causes PEBS assist

nop

are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

The PEBS record format is shown in Table 21-4, and each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 21-4. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	58H	R9
08H	R/EIP	60H	R10
10H	R/EAX	68H	R11
18H	R/EBX	70H	R12
20H	R/ECX	78H	R13
28H	R/EDX	80H	R14
30H	R/ESI	88H	R15
38H	R/EDI	90H	IA32_PERF_GLOBAL_STATUS
40H	R/EBP	98H	Data Linear Address
48H	R/ESP	A0H	Data Source Encoding
50H	R8	A8H	Latency value (core cycles)

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA-32e mode are written to zero.

Bytes AFH:90H are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32_PERF_CAPABILITIES[11:8] encoding of 0001B.

The value written to bytes 97H:90H is the state of the IA32_PERF_GLOBAL_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 21-88. In addition to using IA32_PERFEVTSELx to specify event unit/mask settings and setting the EN_PMCx bit in the IA32_PEBS_ENABLE register for the respective counter, the software must also initialize the DS_BUFFER_MANAGEMENT_AREA data structure in memory to support capturing PEBS records for precise events.

The recording of PEBS records may not operate properly if accesses to the linear addresses in the DS buffer management area or in the PEBS buffer (see below) cause page faults, VM exits, or the setting of accessed or dirty flags in the paging structures (ordinary or EPT). For that reason, system software should establish paging structures (both ordinary and EPT) to prevent such occurrences. Implications of this may be that an operating system should allocate this memory from a non-paged pool and that system software cannot do “lazy” page-table entry propagation for these pages. A virtual-machine monitor may choose to allow use of PEBS by guest software only if EPT maps all guest-physical memory as present and read/write.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

The beginning linear address of the DS_BUFFER_MANAGEMENT_AREA data structure must be programmed into the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 21-18.

- **PEBS Buffer Base**: This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. The processor reads this field to determine the base address of the PEBS buffer.
- **PEBS Index**: This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. The processor reads this field to determine the location of the next PEBS record to write to. After a PEBS record has been written, the processor also updates this field with the address of the next PEBS record to be written. The figure above illustrates the state of PEBS Index after the first PEBS record is written.
- **PEBS Absolute Maximum**: This field represents the absolute address of the maximum length of the allocated PEBS buffer plus the starting address of the PEBS buffer. The processor will not write any PEBS record beyond the end of PEBS buffer, when **PEBS Index** equals **PEBS Absolute Maximum**. No signaling is generated when PEBS buffer is full. Software must reset the **PEBS Index** field to the beginning of the PEBS buffer address to continue capturing PEBS records.

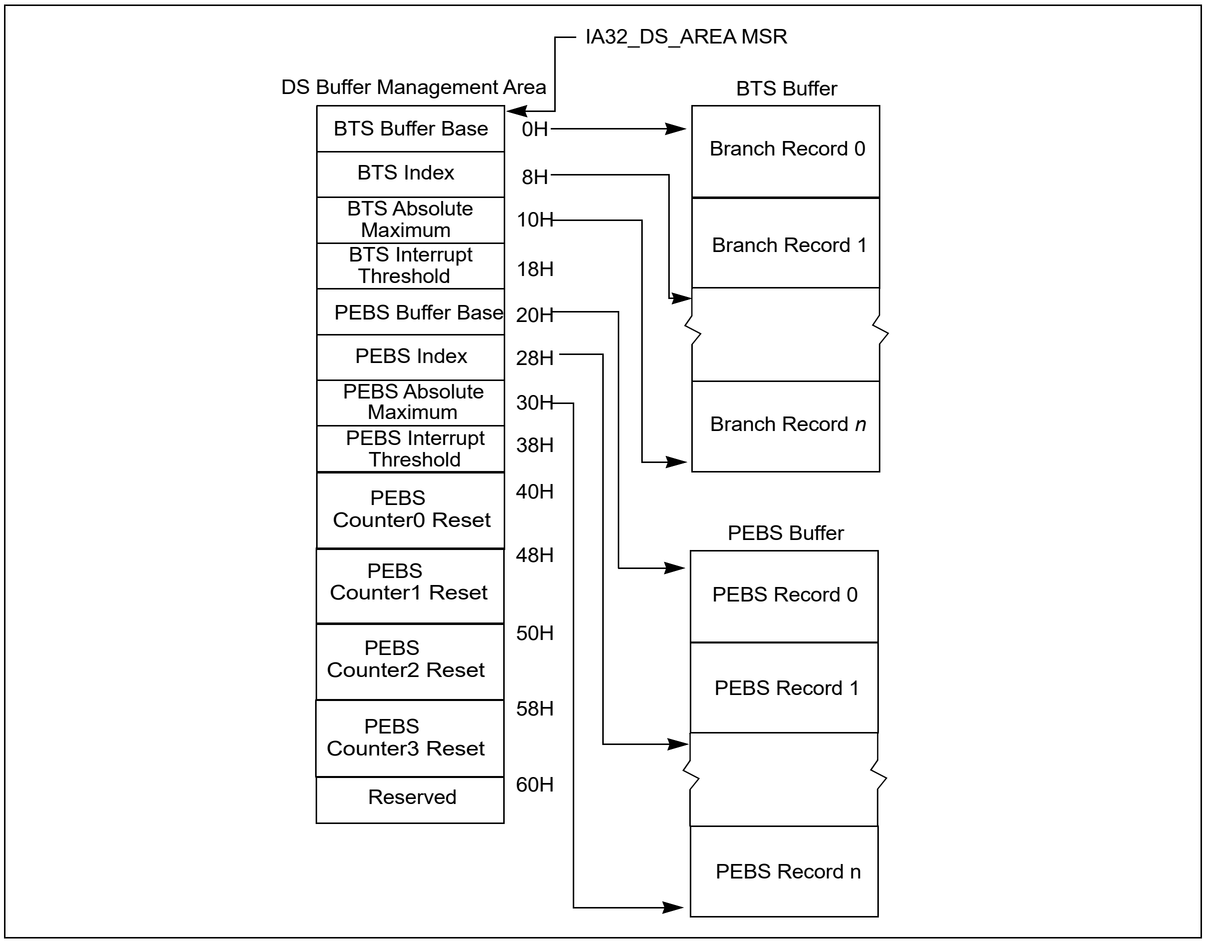


Figure 21-18. PEBS Programming Environment

- **PEBS Interrupt Threshold**: This field specifies the threshold value to trigger a performance interrupt and notify software that the PEBS buffer is nearly full. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After the processor writes a PEBS record and updates **PEBS Index**, if the **PEBS Index** reaches the threshold value of this field, the processor will generate a performance interrupt. This is the same interrupt that is generated by a performance counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When a performance interrupt due to PEBS buffer full is generated, the IA32_PERF_-GLOBAL_STATUS.PEBS_Ovf bit will be set.
- **PEBS CounterX Reset**: This field allows software to set up PEBS counter overflow condition to occur at a rate useful for profiling workload, thereby generating multiple PEBS records to facilitate characterizing the profile the execution of test code. After each PEBS record is written, the processor checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32_PEBS_ENABLED was set). If these conditions are met, then the reset value for each overflowed counter is loaded from the DS Buffer Management Area. For example, if counter IA32_PMC0 caused a PEBS record to be written, then the value of "PEBS Counter 0 Reset" would be written to counter IA32_PMC0. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist.

Performance Counter Prioritization

Performance monitoring interrupts are triggered by a counter transitioning from maximum count to zero (assuming IA32_PerfEvtSelX.INT is set). This same transition will cause PEBS hardware to arm, but not trigger. PEBS hardware triggers upon detection of the first PEBS event after the PEBS hardware has been armed (a 0 to 1 transition of the counter). At this point, a PEBS assist will be undertaken by the processor.

Performance counters (fixed and general-purpose) are prioritized in index order. That is, counter IA32_PMC0 takes precedence over all other counters. Counter IA32_PMC1 takes precedence over counters IA32_PMC2 and IA32_PMC3, and so on. This means that if simultaneous overflows or PEBS assists occur, the appropriate action will be taken for the highest priority performance counter. For example, if IA32_PMC1 cause an overflow interrupt and IA32_PMC2 causes an PEBS assist simultaneously, then the overflow interrupt will be serviced first.

The PEBS threshold interrupt is triggered by the PEBS assist, and is by definition prioritized lower than the PEBS assist. Hardware will not generate separate interrupts for each counter that simultaneously overflows. General-purpose performance counters are prioritized over fixed counters.

If a counter is programmed with a precise (PEBS-enabled) event and programmed to generate a counter overflow interrupt, the PEBS assist is serviced before the counter overflow interrupt is serviced. If in addition the PEBS interrupt threshold is met, the

threshold interrupt is generated after the PEBS assist completes, followed by the counter overflow interrupt (two separate interrupts are generated).

Uncore counters may be programmed to interrupt one or more processor cores (see Section 21.3.1.2). It is possible for interrupts posted from the uncore facility to occur coincident with counter overflow interrupts from the processor core. Software must check core and uncore status registers to determine the exact origin of counter overflow interrupts.

21.3.1.1.2 Load Latency Performance Monitoring Facility

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 21-4. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_INST_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 100H). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001_00000001H.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 21-4, bytes AFH:98H) consists of:

- **Data Linear Address**: This is the linear address of the target of the load operation.
- **Latency Value**: This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.

- **Data Source:** The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 21-5. In the descriptions, local memory refers to system memory physically attached to a processor package, and remote memory refers to system memory physically attached to another processor package.

Table 21-5. Data Source Encoding for Load Latency Record

Encoding	Description
00H	Unknown L3 cache miss.
01H	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
02H	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
03H	This data request was satisfied by the L2.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
06H	L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found.
07H[1]	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and were serviced by another core with a cross core snoop where modified copies were found.
08H	Reserved/L3 MISS. Local homed requests that missed the L3 cache and were serviced by forwarded data following a cross package snoop where no modified copies were found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to exclusive state).
0EH	I/O, Request of input/output operation.
0FH	The request was to uncacheable memory.

NOTES:

1. Bit 7 is supported only for processors with a CPUID DisplayFamily_DisplayModel signature of 06_2A, and 06_2E; otherwise it is reserved.

The layout of MSR_PEBS_LD_LAT_THRESHOLD is shown in Figure 21-19.

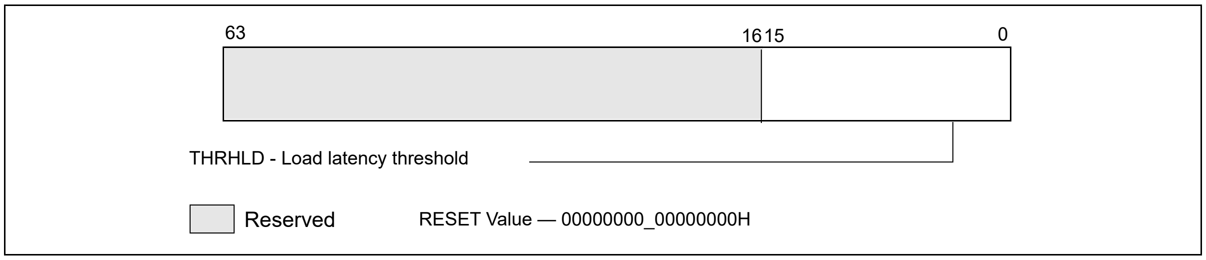

63
16 15
0
THRHLD - Load latency threshold
Reserved
RESET Value — 00000000_00000000H

Figure 21-19. Layout of MSR_PEBS_LD_LAT MSR

Bits 15:0 specifies the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32_PMCx and their latency information is reported in the PEBS record. Otherwise, they are ignored. The minimum value that may be programmed in this field is 3.

21.3.1.1.3 Off-core Response Performance Monitoring in the Processor Core

Programming a performance event using the off-core response facility can choose any of the four IA32_PERFEVT-SELx MSR with specific event codes and predefine mask bit value. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_0. There is only one off-core response configuration MSR. Table 21-6 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 21-6. Off-Core Response Event Encoding

Event code in IA32_PERFEVTSELx	**Mask Value in** IA32_PERFEVTSELx	**Required Off-core Response MSR**
B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)

The layout of MSR_OFFCORE_RSP_0 is shown in Figure 21-20. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.

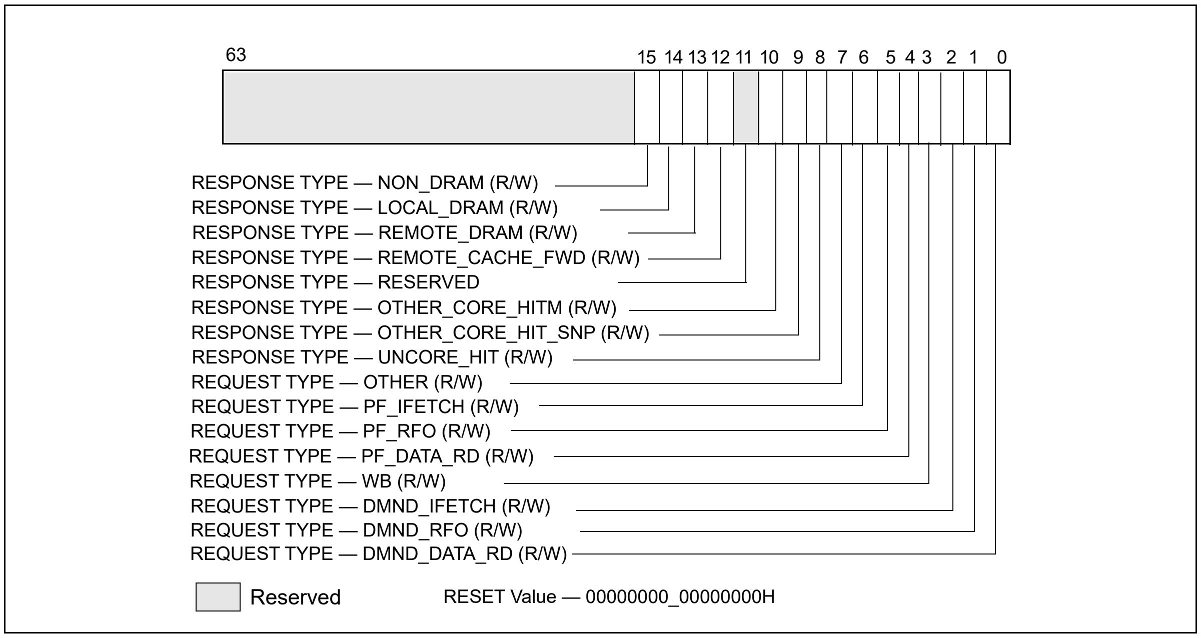

63
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
RESPONSE TYPE — NON_DRAM (R/W)
RESPONSE TYPE — LOCAL_DRAM (R/W)
RESPONSE TYPE — REMOTE_DRAM (R/W)
RESPONSE TYPE — REMOTE_CACHE_FWD (R/W)
RESPONSE TYPE — RESERVED
RESPONSE TYPE — OTHER_CORE_HITM (R/W)
RESPONSE TYPE — OTHER_CORE_HIT_SNP (R/W)
RESPONSE TYPE — UNCORE_HIT (R/W)
REQUEST TYPE — OTHER (R/W)
REQUEST TYPE — PF_IFETCH (R/W)
REQUEST TYPE — PF_RFO (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — WB (R/W)
REQUEST TYPE — DMND_IFETCH (R/W)
REQUEST TYPE — DMND_RFO (R/W)
REQUEST TYPE — DMND_DATA_RD (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 21-20. Layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 to Configure Off-core Response Events

Table 21-7. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.

Table 21-7. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition (Contd.)

Bit Name	Offset	Description
OTHER	7	Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HIT_SNP	9	L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).
OTHER_CORE_HITM	10	L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_FWD	12	L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	Non-DRAM requests that were serviced by IOH.

21.3.1.2 Performance Monitoring Facility in the Uncore

The "uncore" in Nehalem microarchitecture refers to subsystems in the physical processor package that are shared by multiple processor cores. Some of the sub-systems in the uncore include the L3 cache, Intel QuickPath Interconnect link logic, and integrated memory controller. The performance monitoring facilities inside the uncore operates in the same clock domain as the uncore (U-clock domain), which is usually different from the processor core clock domain. The uncore performance monitoring facilities described in this section apply to Intel Xeon processor 5500 series and processors with the following CPUID signatures: 06_1AH, 06_1EH, 06_1FH (see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4). An overview of the uncore performance monitoring facilities is described separately.

The performance monitoring facilities available in the U-clock domain consist of:

- Eight General-purpose counters (MSR_UNCORE_PerfCntr0 through MSR_UNCORE_PerfCntr7). The counters are 48 bits wide. Each counter is associated with a configuration MSR, MSR_UNCORE_PerfEvtSelx, to specify event code, event mask and other event qualification fields. A set of global uncore performance counter enabling/overflow/status control MSRs are also provided for software.
- Performance monitoring in the uncore provides an address/opcode match MSR that provides event qualification control based on address value or QPI command opcode.
- One fixed-function counter, MSR_UNCORE_FixedCntr0. The fixed-function uncore counter increments at the rate of the U-clock when enabled.

  The frequency of the uncore clock domain can be determined from the uncore clock ratio which is available in the PCI configuration space register at offset C0H under device number 0 and Function 0.

21.3.1.2.1 Uncore Performance Monitoring Management Facility

MSR_UNCORE_PERF_GLOBAL_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 21-21 shows the layout of MSR_UNCORE_PERF_GLOBAL_CTRL for an uncore that is shared by four processor cores in a physical package.

- EN_PCn (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR_UNCORE_PerfCntr n.
- EN_FC0 (bit 32): When set, enables counting for the fixed-function uncore counter MSR_UNCORE_FixedCntr0.
- EN_PMI_COREn (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32_DEBUGCTL.Offcore_PMI_EN to 1.

- PMI_FRZ (bit 63): When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in MSR_UNCORE_PERF_GLOBAL_CTRL upon exit from the ISR.

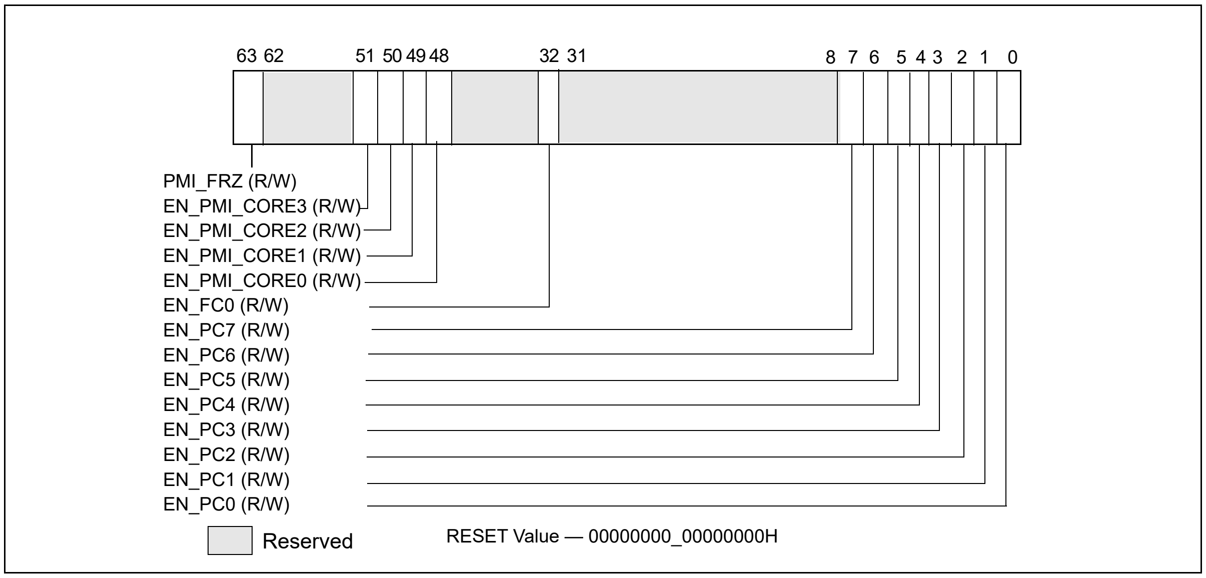


Figure 21-21. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR

MSR_UNCORE_PERF_GLOBAL_STATUS provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 21-22 shows the layout of MSR_UNCORE_PERF_GLOBAL_STATUS.

- OVF_PCn (bit n, n = 0, 7): When set, indicates general-purpose uncore counter MSR_UNCORE_PerfCntr n has overflowed.
- OVF_FC0 (bit 32): When set, indicates the fixed-function uncore counter MSR_UNCORE_FixedCntr0 has overflowed.
- OVF_PMI (bit 61): When set indicates that an uncore counter overflowed and generated an interrupt request.
- CHG (bit 63): When set indicates that at least one status bit in MSR_UNCORE_PERF_GLOBAL_STATUS register has changed state.

MSR_UNCORE_PERF_GLOBAL_OVF_CTRL allows software to clear the status bits in the UNCORE_PERF_-GLOBAL_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.

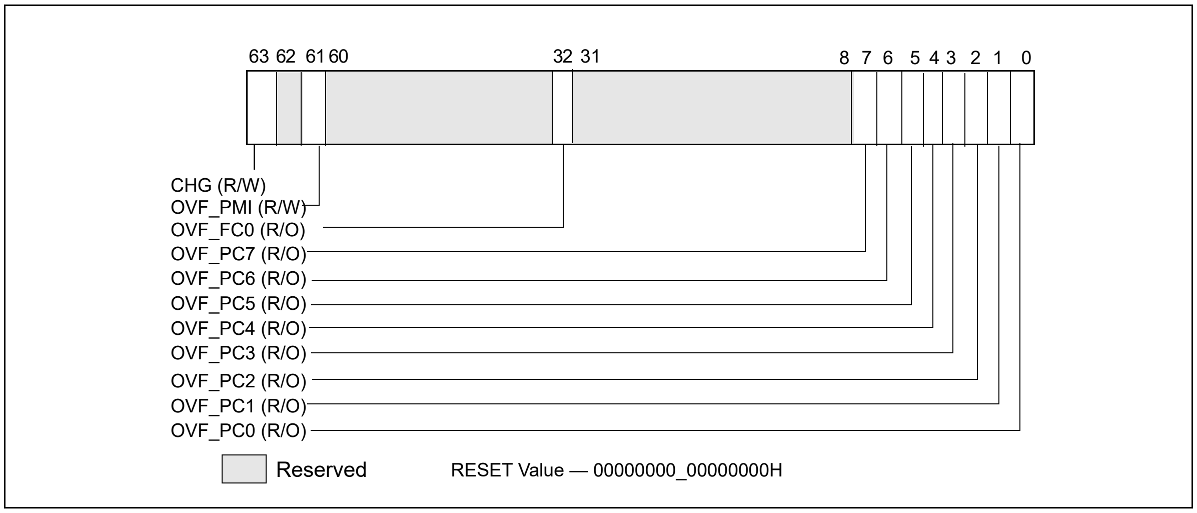


Figure 21-22. Layout of MSR_UNCORE_PERF_GLOBAL_STATUS MSR

Figure 21-23 shows the layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL.

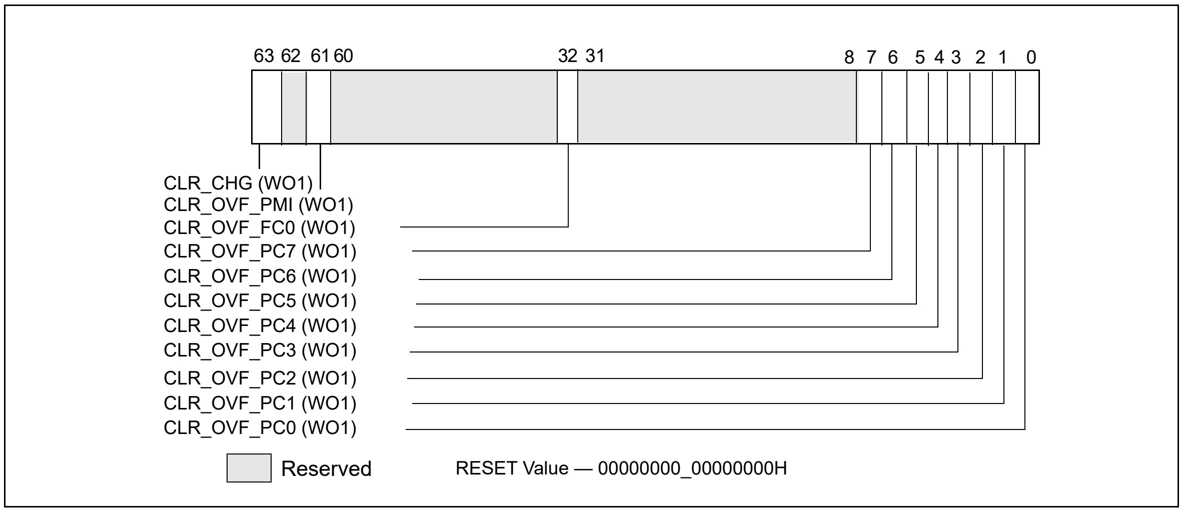


Figure 21-23. Layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL MSR

- CLR_OVF_PCn (bit n, n = 0, 7): Set this bit to clear the overflow status for general-purpose uncore counter MSR_UNCORE_PerfCntr n. Writing a value other than 1 is ignored.
- CLR_OVF_FC0 (bit 32): Set this bit to clear the overflow status for the fixed-function uncore counter MSR_UNCORE_FixedCntr0. Writing a value other than 1 is ignored.
- CLR_OVF_PMI (bit 61): Set this bit to clear the OVF_PMI flag in MSR_UNCORE_PERF_GLOBAL_STATUS. Writing a value other than 1 is ignored.
- CLR_CHG (bit 63): Set this bit to clear the CHG flag in MSR_UNCORE_PERF_GLOBAL_STATUS register. Writing a value other than 1 is ignored.

21.3.1.2.2 Uncore Performance Event Configuration Facility

MSR_UNCORE_PerfEvtSel0 through MSR_UNCORE_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corresponding MSR_UNCORE_PerfEvtSelx and counting must be enabled using the respective EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL. Figure 21-24 shows the layout of MSR_UNCORE_PERFEVTSELx.

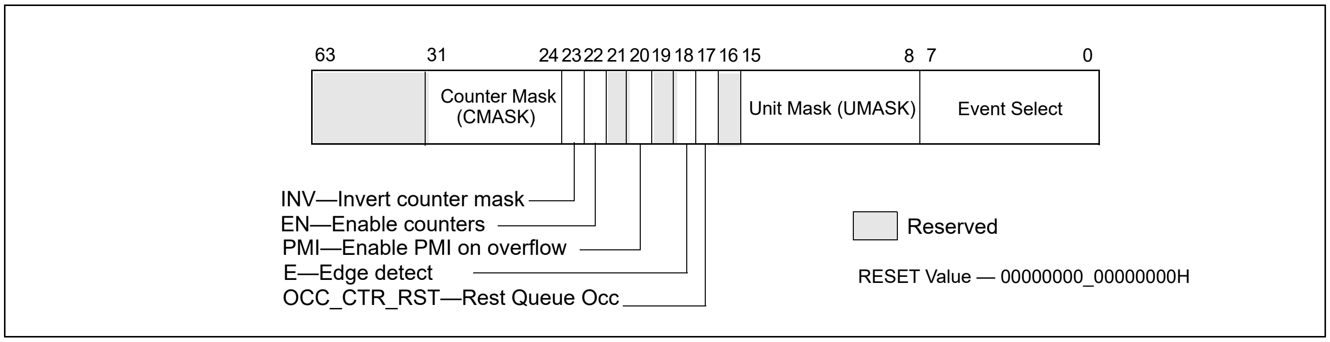


Figure 21-24. Layout of MSR_UNCORE_PERFEVTSELx MSRs

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8) : Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC_CTR_RST (bit17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 21-25 shows the layout of MSR_UNCORE_FIXED_CTR_CTRL.

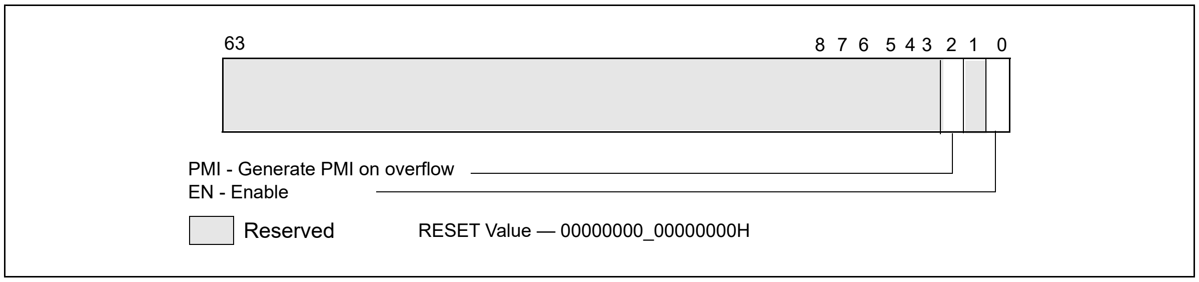

63
8
7
6
5
4
3
2
1
0
PMI - Generate PMI on overflow
EN - Enable
Reserved
RESET Value — 00000000_00000000H

Figure 21-25. Layout of MSR_UNCORE_FIXED_CTR_CTRL MSR

- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN_FC0 bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.

Both the general-purpose counters (MSR_UNCORE_PerfCntr) and the fixed-function counter (MSR_UNCORE_-FixedCntr0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

21.3.1.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR_UNCORE_PERFEVTSELx is used to select different uncore event logic unit. When the event "ADDR_OPCODE_MATCH" is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR_UNCORE_ADDR_OPCODE_MATCH register. The layout is shown in Figure 21-26.

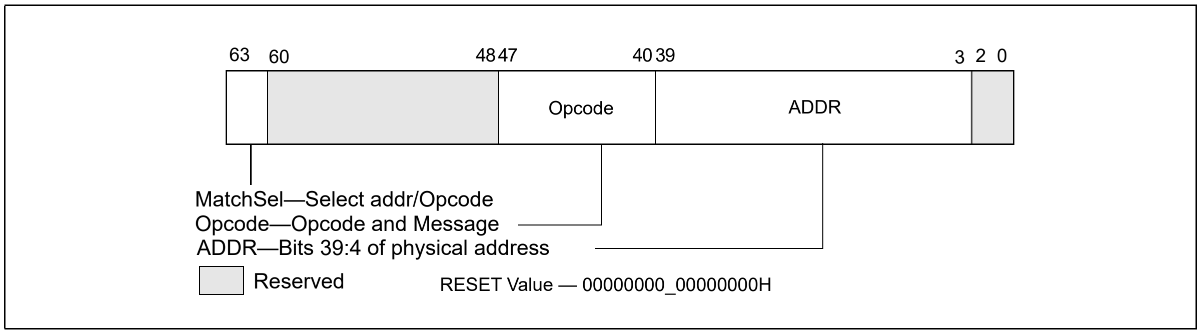

63
60
48
47
40
39
3
2
0
Opcode
ADDR
MatchSel—Select addr/Opcode
Opcode—Opcode and Message
ADDR—Bits 39:4 of physical address
Reserved
RESET Value — 00000000_00000000H

Figure 21-26. Layout of MSR_UNCORE_ADDR_OPCODE_MATCH MSR

- Addr (bits 39:3): The physical address to match if "MatchSel" field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- Opcode (bits 47:40) : Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - — Bits 43:40 specify the QPI packet header opcode.
  - — Bits 47:44 specify the QPI message classes.

  Table 21-8 lists the encodings supported in the opcode field.

Table 21-8. Opcode Field Encoding for MSR_UNCORE_ADDR_OPCODE_MATCH

Opcode [43:40]	QPI Message Class		
	Home Request [47:44] = 0000B	Snoop Response [47:44] = 0001B	Data Response [47:44] = 1110B
		1	
DMND_IFETCH	2	2	
WB	3	3	
PF_DATA_RD	4	4	
PF_RFO	5	5	
PF_IFETCH	6	6	
OTHER	7	7	
NON_DRAM	15	15	

- MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  - 000B: Disable addr_opcode match hardware.
  - 100B: Count if only the address field matches.
  - 010B: Count if only the opcode field matches.
  - 110B: Count if either opcode field matches or the address field matches.
  - 001B: Count only if both opcode and address field match.
  - Other encoding are reserved.

21.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 21-27.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

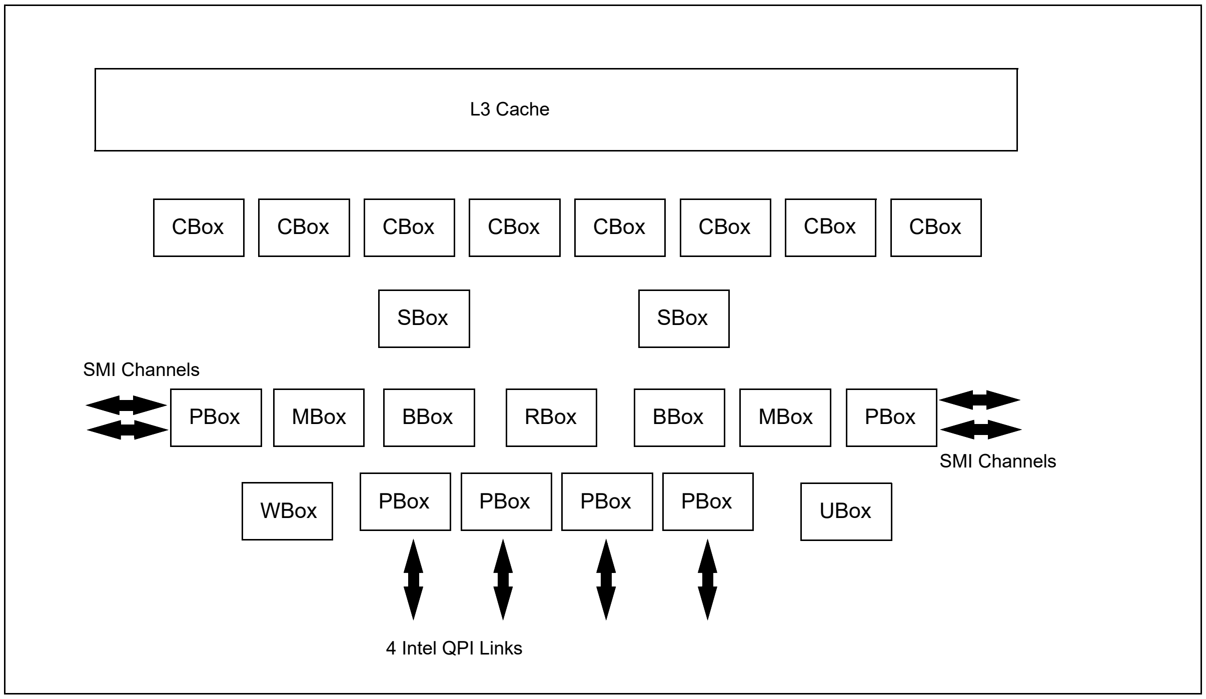


Figure 21-27. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 21-9 summarizes the number MSRs for uncore PMU for each box.

Table 21-9. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 (2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar to the "global control" programming interface, IA32_PERF_GLOBAL_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR_U_PMON_GLOBAL_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual MSRs that provide uncore PMU interfaces are listed in Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, Table 2-17 under the general naming style of MSR_%box#%_PMON_%scope_function%, where %box#% designates the type of box and zero-based index if there are more the one box of the same type, %scope_function% follows the examples below:

- Multi-counter enabling MSRs: MSR_U_PMON_GLOBAL_CTL, MSR_S0_PMON_BOX_CTL, MSR_C7_PMON_BOX_CTL, etc.
- Multi-counter status MSRs: MSR_U_PMON_GLOBAL_STATUS, MSR_S0_PMON_BOX_STATUS, MSR_C7_PMON_BOX_STATUS, etc.
- Multi-counter overflow control MSRs: MSR_U_PMON_GLOBAL_OVF_CTL, MSR_S0_PMON_BOX_OVF_CTL, MSR_C7_PMON_BOX_OVF_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g., MSR_U_PMON_CTR, MSR_S0_PMON_CTR0, MSR_C7_PMON_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g., MSR_U_PMON_EVNT_SEL, MSR_S0_PMON_EVNT_SEL0, MSR_C7_PMON_EVNT_SEL5, etc.
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g., MSR_M0_PMON_TIMESTAMP, MSR_R0_PMON_IPERF0_P1, MSR_S1_PMON_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document "Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide".

21.3.2 Performance Monitoring for Processors Based on Westmere Microarchitecture

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 21.6.3 also apply to processors based on Westmere microarchitecture.

Table 21-6 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR_OFFCORE_RSP_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore response event using event code 0BBH and MSR_OFFCORE_RSP_1 (address 1A7H) are supported in processors based on Westmere microarchitecture. The event code and event mask definitions of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

The load latency facility is the same as described in Section 21.3.1.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB_MISS and LOCK, see Table 21-14.

21.3.3 Intel® Xeon® Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel® Xeon® processor E7 family is the same as those supported in the Intel Xeon processor 5600 series[1]. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore subsystem is similar to that shown in Figure 21-27, with the additional capability that up to 10 C-Box units are supported.

Table 21-10 summarizes the number MSRs for uncore PMU for each box.

Table 21-10. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	10	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 (2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E7 family is available in the "Intel® Xeon® Processor E7 Uncore Performance Monitoring Programming Reference Manual".

21.3.4 Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture

Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel® Xeon® processor E3-1200 family are based on Sandy Bridge microarchitecture; this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 21.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 21.2.3.

The core PMU's capability is similar to those described in Section 21.3.1.1 and Section 21.6.3, with some differences and enhancements relative to Westmere microarchitecture summarized in Table 21-11.

1. Exceptions are indicated for event code 0FH in the event list for this processor (https://perfmon-events.intel.com/); and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 21-14.

Table 21-11. Core PMU Comparison

Box	Sandy Bridge Microarchitecture	Westmere Microarchitecture	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 21.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 21.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 21.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to determine # of counters. See Section 21.2.1.
PMI Overhead Mitigation	• Freeze_PerfMon_on_PMI with legacy semantics. • Freeze_LBR_on_PMI with legacy semantics for branch profiling. • Freeze_while_SMM.	• Freeze_PerfMon_on_PMI with legacy semantics. • Freeze_LBR_on_PMI with legacy semantics for branch profiling. • Freeze_while_SMM.	See Section 19.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 21-13.	See Table 21-88.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 21.3.4.4.2; • Data source encoding • STLB miss encoding • Lock transaction encoding	Data source encoding	
PEBS-Precise Store	Section 21.3.4.4.3	No	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL).	No	
Off-core Response Event	MSR 1A6H and 1A7H, extended request and response types.	MSR 1A6H and 1A7H, limited response types.	Nehalem supports 1A6H only.

21.3.4.1 Global Counter Control Facilities in Sandy Bridge Microarchitecture

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Sandy Bridge microarchitecture. Software must use CPUID to determine the number performance counters/event select registers (See Section).

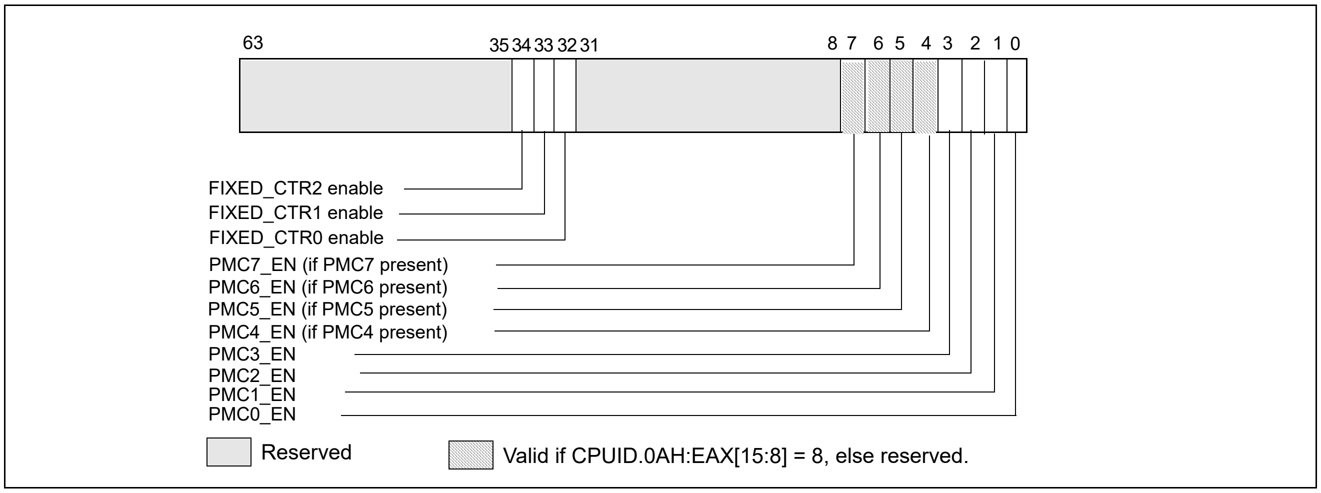

63
35 34 33 32 31
8 7 6 5 4 3 2 1 0
FIXED_CTR2 enable
FIXED_CTR1 enable
FIXED_CTR0 enable
PMC7_EN (if PMC7 present)
PMC6_EN (if PMC6 present)
PMC5_EN (if PMC5 present)
PMC4_EN (if PMC4 present)
PMC3_EN
PMC2_EN
PMC1_EN
PMC0_EN
Reserved
Valid if CPUID.0AH:EAX[15:8] = 8, else reserved.

Figure 21-28. IA32_PERF_GLOBAL_CTRL MSR in Sandy Bridge Microarchitecture

Figure 21-46 depicts the layout of IA32_PERF_GLOBAL_CTRL MSR. The enable bits (PMC4_EN, PMC5_EN, PMC6_EN, PMC7_EN) corresponding to IA32_PMC4-IA32_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of '8'. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32_PERF_GLOBAL_CTRL is ANDed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the ANDed results is true; counting is disabled when the result is false.
IA32_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer (see Figure 21-29). A value of 1 in each bit of the PMCx_OVF field indicates an overflow condition has occurred in the associated counter.

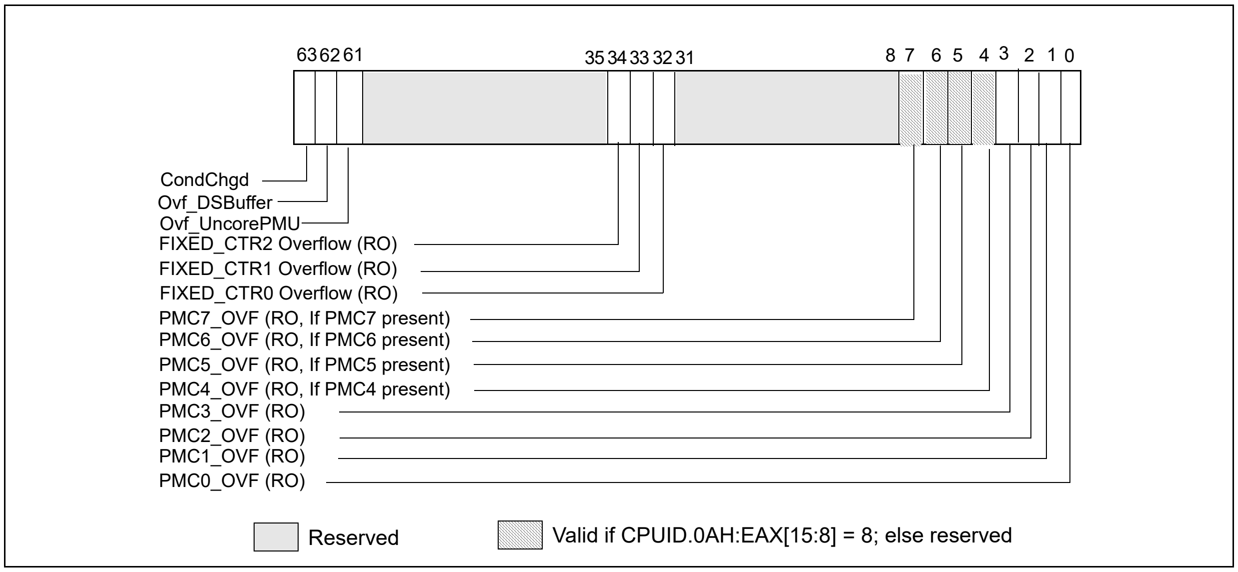


Figure 21-29. IA32_PERF_GLOBAL_STATUS MSR in Sandy Bridge Microarchitecture

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 19.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 21-30). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt based sampling.
- Reloading counter values to continue sampling.
- Disabling event counting or interrupt based sampling.

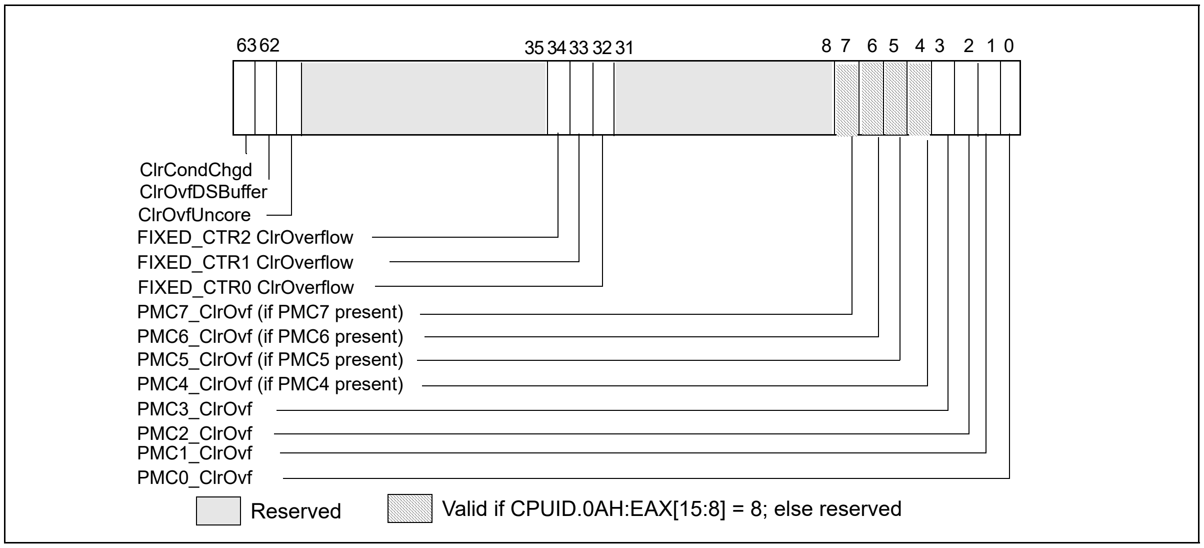


Figure 21-30. IA32_PERF_GLOBAL_OVF_CTRL MSR in Sandy Bridge Microarchitecture

21.3.4.2 Counter Coalescence

In processors based on Sandy Bridge microarchitecture, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report the number of counters visible to software.

If a processor core is shared by two logical processors, each logical processors can access up to four counters (IA32_PMC0-IA32_PMC3). This is the same as in the prior generation for processors based on Nehalem microarchitecture.

If a processor core is not shared by two logical processors, up to eight general-purpose counters are visible. If CPUID.0AH:EAX[15:8] reports 8 counters, then IA32_PMC4-IA32_PMC7 would occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32_PERFEVTSEL4-IA32_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32_PMC4-IA32_PMC7, IA32_PMC4-IA32_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, or IA32_PERF_-GLOBAL_OVF_CTL will also cause #GP.

21.3.4.3 Full Width Writes to Performance Counters

Processors based on Sandy Bridge microarchitecture support full-width writes to the general-purpose counters, IA32_PMCx. Support of full-width writes are enumerated by IA32_PERF_CAPABILITIES.FW_WRITES[13] (see Section 21.2.4).

The default behavior of IA32_PMCx is unchanged, i.e., WRMSR to IA32_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32_A_PMCx by testing IA32_PERF_CAPABILITIES[13].

21.3.4.4 PEBS Support in Sandy Bridge Microarchitecture

Processors based on Sandy Bridge microarchitecture support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Westmere microarchitecture is summarized in Table 21-12.

Table 21-12. PEBS Facility Comparison

Box	Sandy Bridge Microarchitecture	Westmere Microarchitecture	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 21.3.1.1.1	Section 21.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 21-31	Figure 21-17	
PEBS record layout	Physical Layout same as Table 21-4.	Table 21-4	Enhanced fields at offsets 98H, A0H, A8H.
PEBS Events	See Table 21-13.	See Table 21-88.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 21-14.	Table 21-5	
PEBS-Precise Store	Yes; see Section 21.3.4.4.3.	No	IA32_PMC3 only
PEBS-PDIR	Yes	No	IA32_PMC1 only
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

In IA32_PEBS_ENABLE MSR, bit 63 is defined as PS_ENABLE: When set, this enables IA32_PMC3 to capture precise store information. Only IA32_PMC3 supports the precise store facility. In typical usage of PEBS, the bit fields in IA32_PEBS_ENABLE are written to when the agent software starts PEBS operation; the enabled bit fields should be modified only when re-programming another PEBS event or cleared when the agent uses the performance counters for non-PEBS operations.

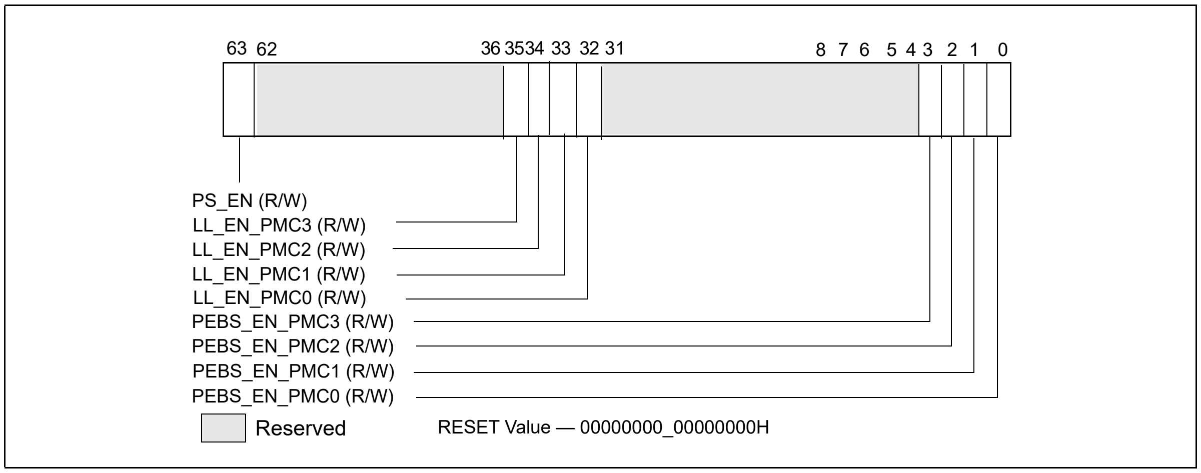


Figure 21-31. Layout of IA32_PEBS_ENABLE MSR

21.3.4.4.1 PEBS Record Format

The layout of PEBS records physically identical to those shown in Table 21-4, but the fields at offsets 98H, A0H, and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H): When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 21-5. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32_PERF_GLOBAL_STATUS corresponding to those counters that both overflowed and were enabled in the IA32_PEBS_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Sandy Bridge microarchitecture is shown in Table 21-13.

Table 21-13. PEBS Performance Events for Sandy Bridge Microarchitecture

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST	01H[1]
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Near_Taken	20H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H

Table 21-13. PEBS Performance Events for Sandy Bridge Microarchitecture (Contd.)

Event Name	Event Select	Sub-event	UMask
MEM_UOPS_RETIRED	D0H	STLB_MISS_LOADS	11H
		STLB_MISS_STORE	12H
		LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H

NOTES:
1. Only available on IA32_PMC1.

21.3.4.4.2 Load Latency Performance Monitoring Facility

The load latency facility in Sandy Bridge microarchitecture is similar to that in prior microarchitectures. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 21-4 and Section 21.3.4.4.1. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_TRANS_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001.00000001H.
- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM_TRANS_RETIRED event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 21-4. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.

Table 21-14. Layout of Data Source Field of Load Latency Record

Field	Position	Description
Source	3:0	See Table 21-5
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

The layout of MSR_PEBS_LD_LAT_THRESHOLD is the same as shown in Figure 21-19.

21.3.4.4.3 Precise Store Facility

Processors based on Sandy Bridge microarchitecture offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.
- Program the MEM_TRANS_RETIRED.PRECISE_STORE event in IA32_PERFEVTSEL3. Only counter 3 (IA32_PMC3) supports collection of precise store information.
- Set IA32_PEBS_ENABLE[3] and IA32_PEBS_ENABLE[63]. This enables IA32_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offsets 98H, A0H, and A8H of Table 21-4. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 21-15. Layout of Precise Store Information In PEBS Record

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	**L1D Hit** (Bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. **STLB Miss** (bit 4): The store missed the STLB if set, otherwise the store hit the STLB **Locked Access** (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.
Reserved	A8H	Reserved

21.3.4.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable "skids" in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Sandy Bridge microarchitecture include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the "skid" problem by providing an early indication of when the INST_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow. On processors based on Sandy Bridge microarchitecture, skid is significantly reduced and can be as little as one instruction. On future implementations, PDIR may eliminate skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and processors based on Sandy Bridge microarchitecture must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily_DisplayModel signatures of 06_2A and 06_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

21.3.4.5 Off-core Response Performance Monitoring

The core PMU in processors based on Sandy Bridge microarchitecture provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Table 21-16 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 21-16. Off-Core Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMC0-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 are shown in Figure 21-32 and Figure 21-33. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

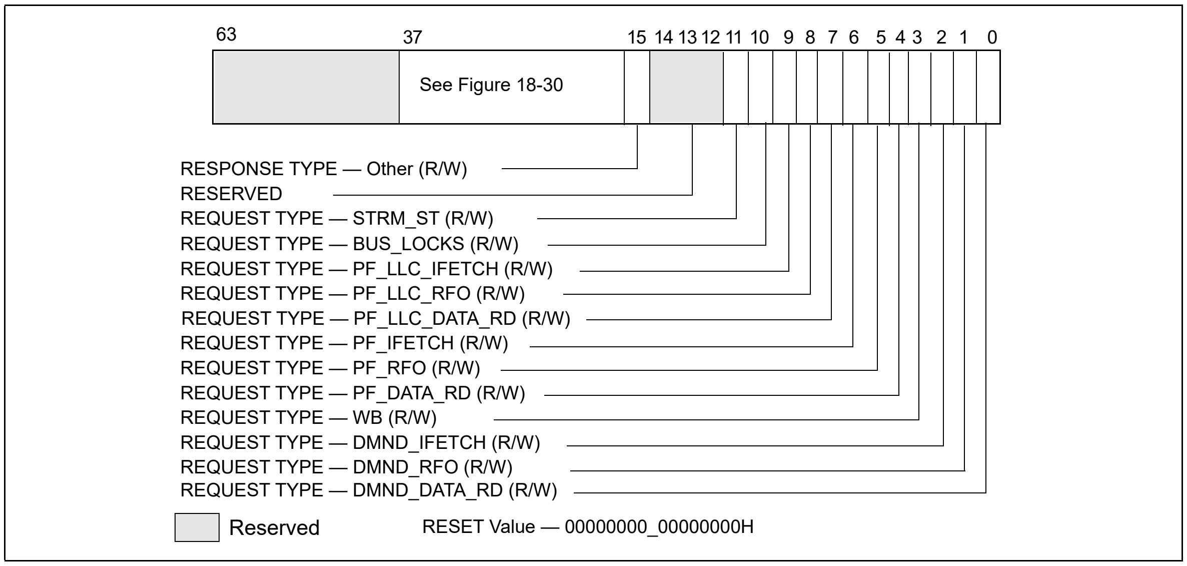

63
37
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
See Figure 18-30
RESPONSE TYPE — Other (R/W)
RESERVED
REQUEST TYPE — STRM_ST (R/W)
REQUEST TYPE — BUS_LOCKS (R/W)
REQUEST TYPE — PF_LLC_IFETCH (R/W)
REQUEST TYPE — PF_LLC_RFO (R/W)
REQUEST TYPE — PF_LLC_DATA_RD (R/W)
REQUEST TYPE — PF_IFETCH (R/W)
REQUEST TYPE — PF_RFO (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — WB (R/W)
REQUEST TYPE — DMND_IFETCH (R/W)
REQUEST TYPE — DMND_RFO (R/W)
REQUEST TYPE — DMND_DATA_RD (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 21-32. Request_Type Fields for MSR_OFFCORE_RSP_x

Table 21-17. MSR_OFFCORE_RSP_x Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests
STRM_ST	11	Streaming store requests
OTHER	15	Any other request that crosses IDI, including I/O.

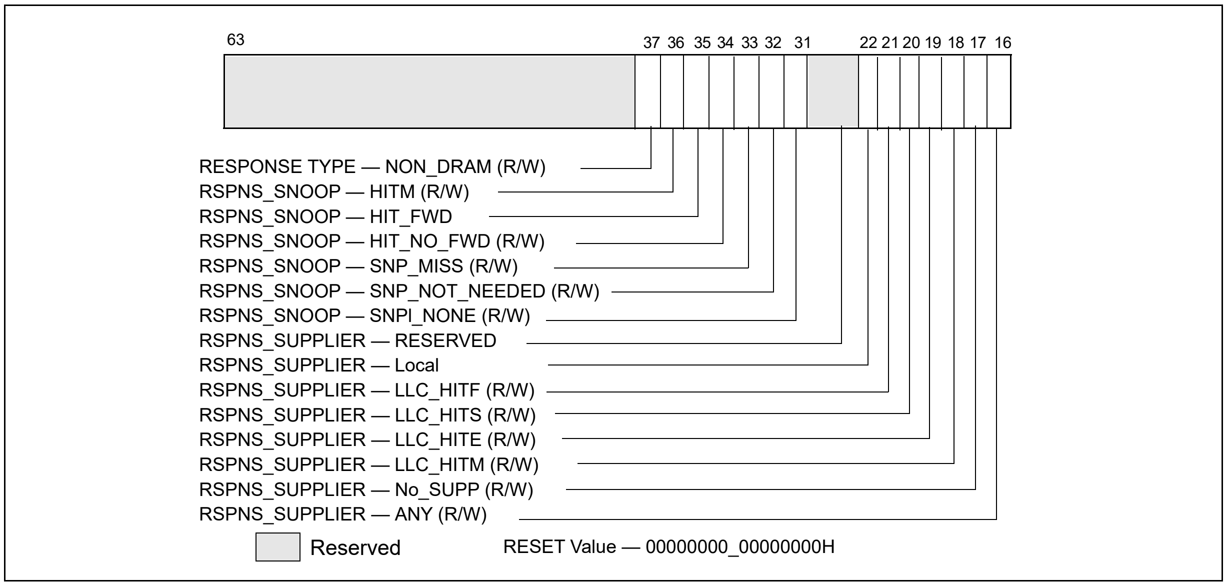

63
37 36 35 34 33 32 31
22 21 20 19 18 17 16
RESPONSE TYPE — NON_DRAM (R/W)
RSPNS_SNOOP — HITM (R/W)
RSPNS_SNOOP — HIT_FWD
RSPNS_SNOOP — HIT_NO_FWD (R/W)
RSPNS_SNOOP — SNP_MISS (R/W)
RSPNS_SNOOP — SNP_NOT_NEEDED (R/W)
RSPNS_SNOOP — SNPl_NONE (R/W)
RSPNS_SUPPLIER — RESERVED
RSPNS_SUPPLIER — Local
RSPNS_SUPPLIER — LLC_HITF (R/W)
RSPNS_SUPPLIER — LLC_HITS (R/W)
RSPNS_SUPPLIER — LLC_HITE (R/W)
RSPNS_SUPPLIER — LLC_HITM (R/W)
RSPNS_SUPPLIER — No_SUPP (R/W)
RSPNS_SUPPLIER — ANY (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 21-33. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSP_x

To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSP_x allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 21-18. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Reserved	30:23	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY” bit is set, the supplier and snoop info bits are ignored.

Table 21-19. MSR_OFFCORE_RSP_x Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	SNP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNP_MISS	33	A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.

21.3.4.6 Uncore Performance Monitoring Facilities in the Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, and Intel® Core™ i3-2xxx Processor Series

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 21.3.1.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 21-34.

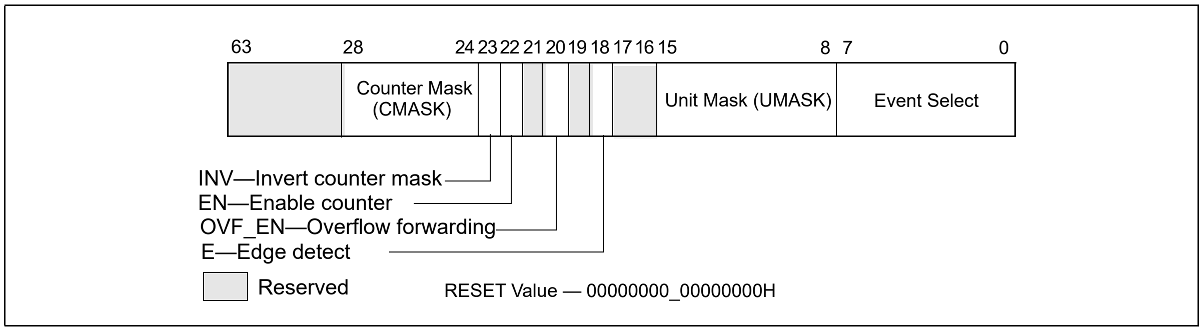


Figure 21-34. Layout of Uncore PERFEVTSEL MSR for a C-Box Unit or the ARB Unit

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see the event list at: https://perfmon-events.intel.com/.
- E (bit 18): Enables edge detection filtering, if 1.
- OVF_EN (bit 20): Enables the overflow indicator from the uncore counter forwarded to MSR_UNC_PERF_-GLOBAL_CTRL, if 1.
- EN (bit 22): Enables the local counter associated with this event select MSR.
- INV (bit 23): Event count increments with non-negative value if 0, with negated value if 1.
- CMASK (bits 28:24): Specifies a positive threshold value to filter raw event count input.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 21-35 shows the layout of the uncore domain global control.

When an uncore counter overflows, a PMI can be routed to a processor core. Bits 3:0 of MSR_UNC_PERF_-GLOBAL_CTRL can be used to select which processor core to handle the uncore PMI. Software must then write to bit 13 of IA32_DEBUGCTL (at address 1D9H) to enable this capability.

- PMI_SEL_Core#: Enables the forwarding of an uncore PMI request to a processor core, if 1. If bit 30 (WakePMI) is '1', a wake request is sent to the respective processor core prior to sending the PMI.
- EN: Enables the fixed uncore counter, the ARB counters, and the CBO counters in the uncore PMU, if 1. This bit is cleared if bit 31 (FREEZE) is set and any enabled uncore counters overflow.
- WakePMI: Controls sending a wake request to any halted processor core before issuing the uncore PMI request. If a processor core was halted and not sent a wake request, the uncore PMI will not be serviced by the processor core.
- FREEZE: Provides the capability to freeze all uncore counters when an overflow condition occurs in a unit counter. When this bit is set, and a counter overflow occurs, the uncore PMU logic will clear the global enable bit (bit 29).

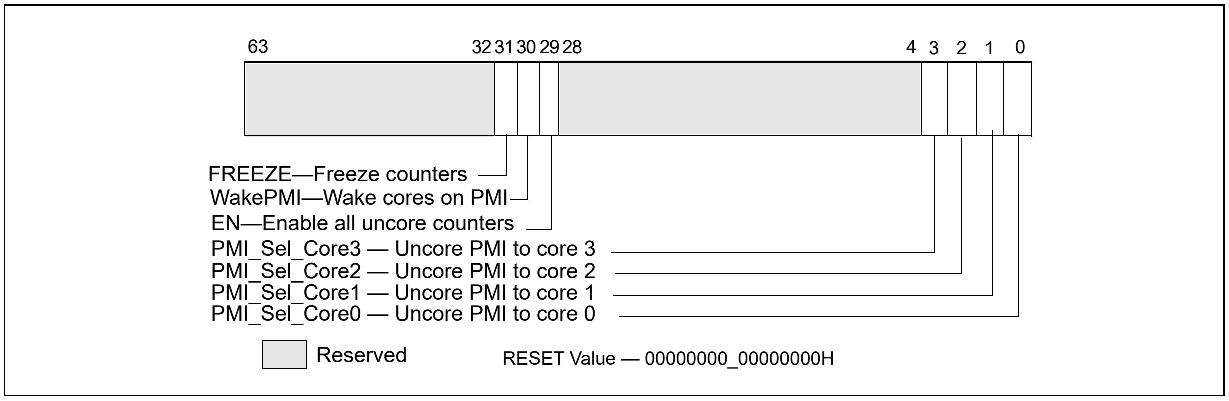

63
32 31 30 29 28
4 3 2 1 0
FREEZE—Freeze counters
WakePMI—Wake cores on PMI
EN—Enable all uncore counters
PMI_Sel_Core3 — Uncore PMI to core 3
PMI_Sel_Core2 — Uncore PMI to core 2
PMI_Sel_Core1 — Uncore PMI to core 1
PMI_Sel_Core0 — Uncore PMI to core 0
Reserved
RESET Value — 00000000_00000000H

Figure 21-35. Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 21-20 summarizes the number MSRs for uncore PMU for each box.

Table 21-20. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-21 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	

Table 21-20. Uncore PMU MSR Summary (Contd.)

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
Fixed Counter	N.A.	N.A.	48	No	Uncore	

21.3.4.6.1 Uncore Performance Monitoring Events

There are certain restrictions on the uncore performance counters in each C-Box. Specifically,

- Occupancy events are supported only with counter 0 but not counter 1.
- Other uncore C-Box events can be programmed with either counter 0 or 1.

The C-Box uncore performance events can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE_RESPONSE and qualified on a per-thread basis.

On the other hand, uncore performance event logic cannot associate its counts with the same level of per-thread qualification attributes as the core PMU events can. Therefore, whenever similar event programming capabilities are available from both core PMU and uncore PMU, the recommendation is that utilizing the core PMU events may be less affected by artifacts, complex interactions and other factors.

21.3.4.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility

The Intel® Xeon® Processor E5 Family (and Intel® Core™ i7-3930K Processor) are based on Sandy Bridge-E microarchitecture. While the processor cores share the same microarchitecture as those of the Intel® Xeon® Processor E3 Family and 2nd generation Intel Core i7-2xxx, Intel Core i5-2xxx, Intel Core i3-2xxx processor series, the uncore subsystems are different. An overview of the uncore performance monitoring facilities of the Intel Xeon processor E5 family (and Intel Core i7-3930K processor) is described in Section 21.3.4.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 21.6.3 through Section 21.3.4.5. However, the MSR_OFFCORE_RSP_0/MSR_OFFCORE_RSP_1 Response Supplier Info field shown in Table 21-18 applies to Intel Core Processors with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2AH; Intel Xeon processor with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2DH supports an additional field for remote DRAM controller shown in Table 21-21. Additionally, there are some small differences in the non-architectural performance monitoring events (see event list available at: https://perfmon-events.intel.com/).

Table 21-21. MSR_OFFCORE_RSP_x Supplier Info Field Definitions

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	LLC_HITM	18	M-state initial lookup stat in L3.
	LLC_HITE	19	E-state
	LLC_HITS	20	S-state
	LLC_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Remote	30:23	Remote DRAM Controller (either all 0s or all 1s).

21.3.4.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5-2600 product family has some similarities with those of the Intel Xeon processor E7 family. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope. For example, each Cbox caching agent has a set of local performance counters, and the power controller unit (PCU) has its own local performance counters. Up to 8 C-Box units are supported in the uncore sub-system.

Table 21-22 summarizes the uncore PMU facilities providing MSR interfaces.

Table 21-22. Uncore PMU MSR Summary for Intel® Xeon® Processor E5 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	4	44	Yes	per-box	None
PCU	1	4	48	Yes	per-box	Match/Mask
U-Box	1	2	44	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 family is available in “Intel® Xeon® Processor E5 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-24.

21.3.5 3rd Generation Intel® Core™ Processor Performance Monitoring Facility

The 3rd generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v2 product family are based on the Ivy Bridge microarchitecture. The performance monitoring facilities in the processor core generally are the same as those described in Section 21.6.3 through Section 21.3.4.5. The non-architectural performance monitoring events supported by the processor core can be found at: https://perfmon-events.intel.com/.

21.3.5.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5 v2 and Intel Xeon Processor E7 v2 product families are based on the Ivy Bridge-E microarchitecture. There are some similarities with those of the Intel Xeon processor E5 family based on the Sandy Bridge microarchitecture. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope.

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v2 and Intel Xeon Processor E7 v2 families are available in the “Intel® Xeon® Processor E5 v2 and E7 v2 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-28.

21.3.6 4th Generation Intel® Core™ Processor Performance Monitoring Facility

The 4th generation Intel® Core™ processor and Intel® Xeon® processor E3-1200 v3 product family are based on the Haswell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 21.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 21.2.3.

The core PMU’s capability is similar to those described in Section 21.6.3 through Section 21.3.4.5, with some differences and enhancements summarized in Table 21-23. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 21.3.6.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® AVX10,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Table 21-23. Core PMU Comparison

Box	Haswell Microarchitecture	Sandy Bridge Microarchitecture	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 21.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 21.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 21.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to determine # of counters. See Section 21.2.1.
PMI Overhead Mitigation	▪ Freeze_PerfMon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_PerfMon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 19.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 21-13 and Section 21.3.6.5.1.	See Table 21-13.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 21.3.4.4.2.	See Section 21.3.4.4.2.	
PEBS-Precise Store	No, replaced by Data Address profiling.	Section 21.3.4.4.3	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL)	Yes (using precise INST_RETIRED.ALL)	
PEBS-EventingIP	Yes	No	
Data Address Profiling	Yes	No	
LBR Profiling	Yes	Yes	
Call Stack Profiling	Yes, see Section 19.11.	No	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; extended request and response types.	MSR 1A6H and 1A7H; extended request and response types.	
Intel TSX support for PerfMon	See Section 21.3.6.5.	No	

21.3.6.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 4th Generation Intel Core processor is similar to those in processors based on Sandy Bridge microarchitecture, with several enhanced features. The key components and differences of PEBS facility relative to Sandy Bridge microarchitecture is summarized in Table 21-24.

Table 21-24. PEBS Facility Comparison

Box	Haswell Microarchitecture	Sandy Bridge Microarchitecture	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7
PEBS Buffer Programming	Section 21.3.1.1.1	Section 21.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 21-17	Figure 21-31	
PEBS record layout	Table 21-25; enhanced fields at offsets 98H, A0H, A8H, B0H.	Table 21-4; enhanced fields at offsets 98H, A0H, A8H.	

Table 21-24. PEBS Facility Comparison

Box	Haswell Microarchitecture	Sandy Bridge Microarchitecture	Comment
Precise Events	See Table 21-13.	See Table 21-13.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 21-14.	Table 21-14	
PEBS-Precise Store	No, replaced by data address profiling.	Yes; see Section 21.3.4.4.3.	
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

21.3.6.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 21-25. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 21-25. PEBS Record Format for 4th Generation Intel Core Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Data Linear Address
40H	R/EBP	A0H	Data Source Encoding
48H	R/ESP	A8H	Latency value (core cycles)
50H	R8	B0H	EventingIP
58H	R9	B8H	TX Abort Information (Section 21.3.6.5.1)

The layout of PEBS records are almost identical to those shown in Table 21-4. Offset B0H is a new field that records the eventing IP address of the retired instruction that triggered the PEBS assist.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 21.3.4.4.2), PDIR (Section 21.3.4.4.4), and the equivalent capability of precise store in prior generation (see Section 21.3.6.3).

In the core PMU of the 4th generation Intel Core processor, load latency facility and PDIR capabilities are unchanged. However, precise store is replaced by an enhanced capability, data address profiling, that is not restricted to store address. Data address profiling also records information in PEBS records at offsets 98H, A0H, and ABH.

21.3.6.3 PEBS Data Address Profiling

The Data Linear Address facility is also abbreviated as DataLA. The facility is a replacement or extension of the precise store facility in previous processor generations. The DataLA facility complements the load latency facility by providing a means to profile load and store memory references in the system, leverages the PEBS facility, and provides additional information about sampled loads and stores. Having precise memory reference events with linear address information for both loads and stores provides information to improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

The DataLA facility in the 4th generation processor supports the following events configured to use PEBS:

Table 21-26. Precise Events That Supports Data Linear Address Profiling

Event Name	Event Name
MEM_UOPS_RETIRED.STLB_MISS_LOADS	MEM_UOPS_RETIRED.STLB_MISS_STORES
MEM_UOPS_RETIRED.LOCK_LOADS	MEM_UOPS_RETIRED.SPLIT_STORES
MEM_UOPS_RETIRED.SPLIT_LOADS	MEM_UOPS_RETIRED.ALL_STORES
MEM_UOPS_RETIRED.ALL_LOADS	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM
MEM_LOAD_UOPS_RETIRED.L1_HIT	MEM_LOAD_UOPS_RETIRED.L2_HIT
MEM_LOAD_UOPS_RETIRED.L3_HIT	MEM_LOAD_UOPS_RETIRED.L1_MISS
MEM_LOAD_UOPS_RETIRED.L2_MISS	MEM_LOAD_UOPS_RETIRED.L3_MISS
MEM_LOAD_UOPS_RETIRED.HIT_LFB	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS
MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM
UOPS_RETIRED.ALL (if load or store is tagged)	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE

DataLA can use any one of the IA32_PMC0-IA32_PMC3 counters. Counter overflows will initiate the generation of PEBS records. Upon counter overflow, hardware captures the linear address and possible other status information of the retiring memory uop. This information is then written to the PEBS record that is subsequently generated.

To enable the DataLA facility, software must complete the following steps. Please note that the DataLA facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture DataLA information.

- Complete the PEBS configuration steps.
- Program an event listed in Table 21-26 using any one of IA32_PERFEVTSEL0-IA32_PERFEVTSEL3.
- Set the corresponding IA32_PEBS_ENABLE.PEBS_EN_CTRx bit. This enables the corresponding IA32_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H, and A8H, as shown in Table 21-27.

Table 21-27. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_UOPS_RETIRED.STLB_MISS_STORES, MEM_UOPS_RETIRED.SPLIT_STORES, MEM_UOPS_RETIRED.ALL_STORES ▪ Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 21-26.
Reserved	A8H	Always zero.

21.3.6.3.1 EventingIP Record

The PEBS record layout for processors based on Haswell microarchitecture adds a new field at offset 0B0H. This is the eventingIP field that records the IP address of the retired instruction that triggered the PEBS assist. The EIP/RIP field at offset 08H records the IP address of the next instruction to be executed following the PEBS assist.

21.3.6.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 21.3.4.5. The event codes are listed in Table 21-16. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 21-28.
- Supplier information (bits 30:16): see Table 21-29.
- Snoop response information (bits 37:31): see Table 21-19.

Table 21-28. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell Microarchitecture)

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
COREWB	3	Counts the number of modified cachelines written back.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PF_L3_DATA_RD	7	Counts the number of data cacheline reads generated by L3 prefetchers.
PF_L3_RFO	8	Counts the number of RFO requests generated by L3 prefetchers.
PF_L3_CODE_RD	9	Counts the number of code reads generated by L3 prefetchers.
SPLIT_LOCK_UC_LOCK	10	Counts the number of lock requests that split across two cachelines or are to UC memory.
STRM_ST	11	Counts the number of streaming store requests electronically.
Reserved	14:12	Reserved

Table 21-28. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell Microarchitecture) (Contd.)

Bit Name	Offset	Description
OTHER	15	Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 21-29. The fields vary across products (according to CPUID signatures) and is noted in the description.

Table 21-29. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_3CH, 06_46H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	LOCAL	22	Local DRAM Controller.
	Reserved	30:23	Reserved

Table 21-30. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_45H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	L4 Cache
	Reserved	30:26	Reserved

21.3.6.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 21-29 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CPUID signature 06_3FH).

Table 21-31. MSR_OFFCORE_RSP_x Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	L3_HITF	21	F-state
	LOCAL	22	Local DRAM Controller.
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

21.3.6.5 Performance Monitoring and Intel® TSX

Chapter 16 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, describes the details of Intel® Transactional Synchronization Extensions (Intel® TSX). This section describes performance monitoring support for Intel TSX.

If a processor supports Intel TSX, the core PMU enhances its IA32_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32_PERFEVTSELx is shown in Figure 21-36. The two additional bit fields are:

- **IN_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32_PERFEVTSEL2.

When the IA32_PERFEVTSELx MSR is programmed with both IN_TX=0 and IN_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN_TXCP bit. See Table 2-29.

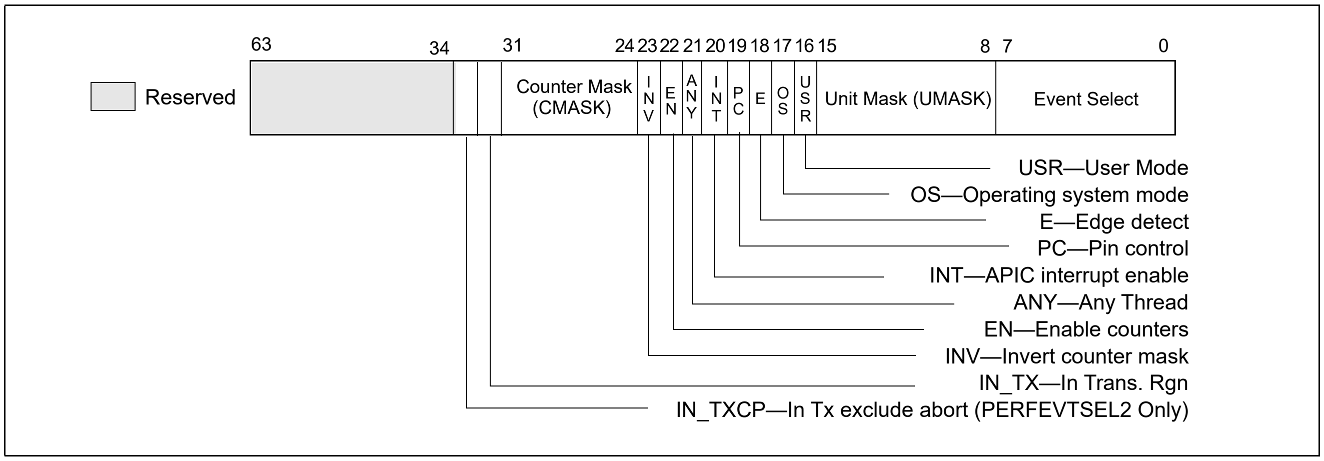


Figure 21-36. Layout of IA32_PERFEVTSELx MSRs Supporting Intel TSX

A common usage of setting IN_TXCP=1 is to capture the number of events that were discarded due to a transactional abort. With IA32_PMC2 configured to count in such a manner, then when a transactional region aborts, the value for that counter is restored to the value it had prior to the aborted transactional region. As a result, any updates performed to the counter during the aborted transactional region are discarded.

On the other hand, setting IN_TX=1 can be used to drill down on the performance characteristics of transactional code regions. When a PMCx is configured with the corresponding IA32_PERFEVTSELx.IN_TX=1, only eventing conditions that occur inside transactional code regions are propagated to the event logic and reflected in the counter result. Eventing conditions specified by IA32_PERFEVTSELx but occurring outside a transactional region are discarded.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they can be found at: https://perfmon-events.intel.com/.

21.3.6.5.1 Intel® TSX and PEBS Support

If a PEBS event would have occurred inside a transactional region, then the transactional region first aborts, and then the PEBS event is processed.

Two of the TSX performance monitoring events also support using the PEBS facility to capture additional information. They are:

- HLE_RETIRED.ABORTED (encoding C8H mask 04H),
- RTM_RETIRED.ABORTED (encoding C9H mask 04H).

A transactional abort (HLE_RETIRED.ABORTED,RTM_RETIRED.ABORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.

Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pended at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 21-32.

Table 21-32. TX Abort Information Field Definition

Bit Name	Offset	Description
Cycles_Last_TX	31:0	The number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
HLE_Abort	32	If set, the abort information corresponds to an aborted HLE execution
RTM_Abort	33	If set, the abort information corresponds to an aborted RTM execution
Instruction_Abort	34	If set, the abort was associated with the instruction corresponding to the eventing IP (offset 0B0H) within the transactional region.
Non_Instruction_Abort	35	If set, the instruction corresponding to the eventing IP may not necessarily be related to the transactional abort.
Retry	36	If set, retrying the transactional execution may have succeeded.
Data_Conflict	37	If set, another logical processor conflicted with a memory address that was part of the transactional region that aborted.
Capacity Writes	38	If set, the transactional region aborted due to exceeding resources for transactional writes.
Capacity Reads	39	If set, the transactional region aborted due to exceeding resources for transactional reads.
In_Suspend	40	Transaction was aborted while in a suspend region. This is an Intel Xeon processor only feature, available beginning with 4th generation Intel Xeon Scalable Processor Family; otherwise reserved.
Reserved	63:41	Reserved

21.3.6.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors

The uncore sub-system in the 4th Generation Intel® Core™ processors provides its own performance monitoring facility. The uncore PMU facility provides dedicated MSRs to select uncore performance monitoring events in a similar manner as those described in Section 21.3.4.6.

The ARB unit and each C-Box provide local pairs of event select MSR and counter register. The layout of the event select MSRs in the C-Boxes are identical as shown in Figure 21-34.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 21-35 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 21-20 summarizes the number MSRs for uncore PMU for each box.

Table 21-33. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-21 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

The uncore performance events for the C-Box and ARB units can be found at: https://perfmon-events.intel.com/.

21.3.6.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in “Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-33.

21.3.7 5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility

The 5th Generation Intel® Core™ processor and the Intel® Core™ M processor families are based on the Broadwell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 21.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 21.2.3.

The core PMU has the same capability as those described in Section 21.3.6. IA32_PERF_GLOBAL_STATUS provide a bit indicator (bit 55) for PMI handler to distinguish PMI due to output buffer overflow condition due to accumulating packet data from Intel Processor Trace.

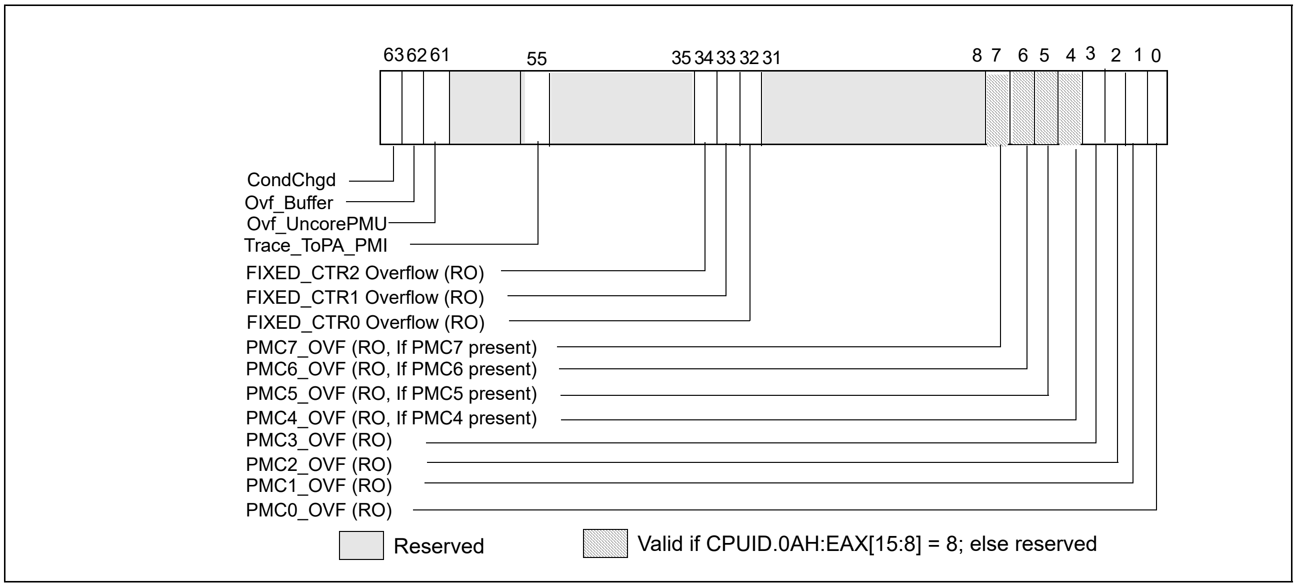


Figure 21-37. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture

Details of Intel Processor Trace is described in Chapter 34, "Intel® Processor Trace." The IA32_PERF_GLOBAL_OVF_CTRL MSR provides a corresponding reset control bit.

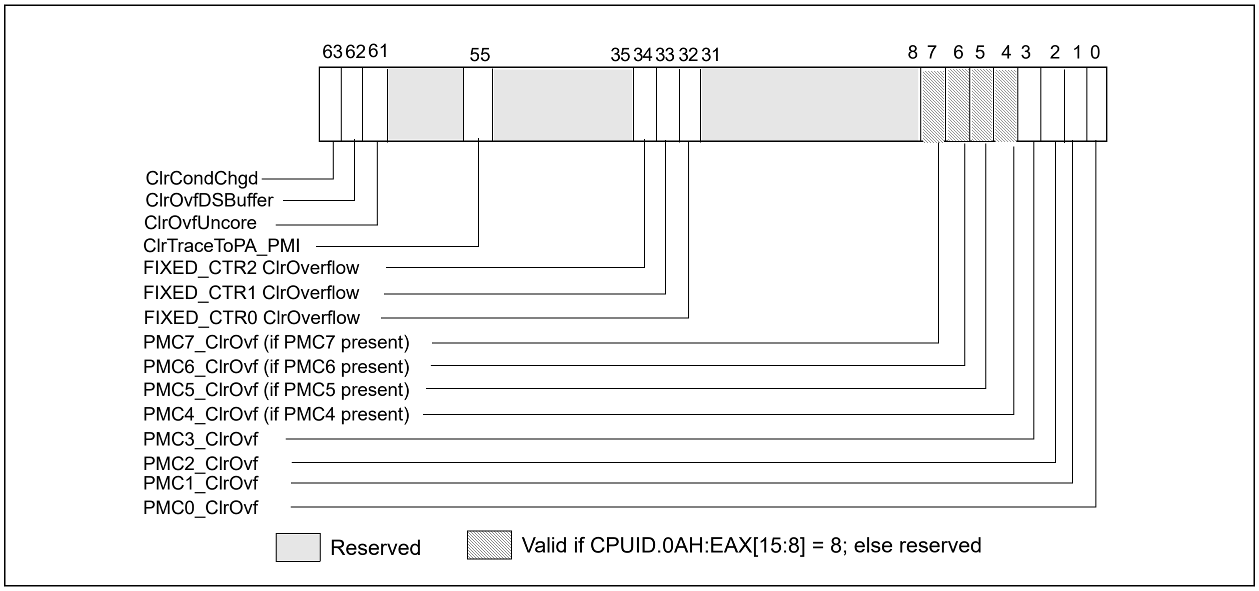


Figure 21-38. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events can be found at: https://perfmon-events.intel.com/.

21.3.8 6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture. For these microarchitectures, the core PMU supports architectural performance monitoring capability with version ID 4 (see Section 21.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 21.2.4.

The core PMU's capability is similar to those described in Section 21.6.3 through Section 21.3.4.5, with some differences and enhancements summarized in Table 21-34. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 21.3.6.5. For details of Intel TSX, see Chapter 16, "Programming with Intel® AVX10," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 41, "Enclave Code Debug and Profiling."

Table 21-34. Core PMU Comparison

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
# of Fixed counters per thread	3	3	Use CPUID to determine # of counters. See Section 21.2.1.
# of general-purpose counters per core	8	8	Use CPUID to determine # of counters. See Section 21.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 21.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to determine # of counters. See Section 21.2.1.
Architectural PerfMon version	4	3	See Section 21.2.4
PMI Overhead Mitigation	▪ Freeze_PerfMon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics. ▪ Freeze_while_SMM.	▪ Freeze_PerfMon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 19.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 21.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz, ASCI	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow (applicable to Broadwell microarchitecture)	See Section 21.2.4.

Table 21-34. Core PMU Comparison (Contd.)

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz ▪ LBR_Frz	NA	See Section 21.2.4.1.
PerfMon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 21.2.4.3.
Precise Events	See Table 21-37.	See Table 21-13.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 21.3.8.2.	No	
LBR Record Format Encoding	000101b	000100b	Section 19.4.8.1
LBR Size	32 entries	16 entries	
LBR Entry	From_IP/To_IP/LBR_Info triplet	From_IP/To_IP pair	Section 19.12
LBR Timing	Yes	No	Section 19.12.1
Call Stack Profiling	Yes, see Section 19.11	Yes, see Section 19.11	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types.	MSR 1A6H and 1A7H; Extended request and response types.	
Intel TSX support for PerfMon	See Section 21.3.6.5.	See Section 21.3.6.5.	

21.3.8.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th generation, 7th generation and 8th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 21-35.

Table 21-35. PEBS Facility Comparison

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 21.3.1.1.1	Section 21.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 21-17	Figure 21-17	
PEBS-EventingIP	Yes	Yes	
PEBS record format encoding	0011b	0010b	
PEBS record layout	Table 21-36; enhanced fields at offsets 98H- B8H; and TSC record field at C0H.	Table 21-25; enhanced fields at offsets 98H, A0H, A8H, B0H.	
Multi-counter PEBS resolution	PEBS record 90H resolves the eventing counter overflow.	PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS.	
Precise Events	See Table 21-37.	See Table 21-13.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 21.3.4.4.2.	See Section 21.3.4.4.2.	
Data Address Profiling	Yes	Yes	

Table 21-35. PEBS Facility Comparison (Contd.)

Box	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Haswell and Broadwell Microarchitectures	Comment
FrontEnd event support	FrontEnd_Retried event and MSR_PEBS_FRONTEND.	No	IA32_PMC0-PMC3 only.

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTES

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

21.3.8.1.1 PEBS Data Format

The PEBS record format for the 6th generation, 7th generation and 8th generation Intel Core processors is reporting with encoding 0011b in IA32_PERF_CAPABILITIES[11:8]. The lay out is shown in Table 21-36. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 21-36. PEBS Record Format for the 6th Generation, 7th Generation, and 8th Generation Intel Core Processor Families

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 21.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 21-25.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 21.3.4.4.2), PDIR (Section 21.3.4.4.4), and data address profiling (Section 21.3.6.3).

In the core PMU of the 6th generation, 7th generation and 8th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th generation and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32_PERF_GLOBAL_STATUS may be useful to resolve the situations when more than one of IA32_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the "applicable counter" field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

21.3.8.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake, Kaby Lake and Coffee Lake microarchitectures is shown in Table 21-37.

Table 21-37. Precise Events for the Skylake, Kaby Lake, and Coffee Lake Microarchitectures

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST[1]	01H
		ALL_CYCLES[2]	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRED	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRED	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRED	C6H	<Programmable[3]>	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED[2]	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRED[4]	D1H	L1_HIT	01H
		L2_HIT	02H
		L3_HIT	04H
		L1_MISS	08H
		L2_MISS	10H
		L3_MISS	20H
		HIT_LFB	40H

Table 21-37. Precise Events for the Skylake, Kaby Lake, and Coffee Lake Microarchitectures (Contd.)

Event Name	Event Select	Sub-event	UMask
MEM_LOAD_L3_HIT_RETIRED[2]	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

NOTES:

1. Only available on IA32_PMC1.
2. INST_RETIRED.ALL_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR_PEBS_FRONTEND, see Section 21.3.8.3
4. Instruction with at least one load uop experiencing the condition specified in the UMask.

21.3.8.1.3 Data Address Profiling

The PEBS Data address profiling on the 6th generation, 7th generation and 8th generation Intel Core processors is largely unchanged from the prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H, and A8H, as shown in Table 21-27.

Table 21-38. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES. ▪ Other bits are zero.
Reserved	A8H	Always zero.

21.3.8.2 Frontend Retired Facility

The Skylake Core PMU has been extended to cover common microarchitectural conditions related to the front end pipeline in addition to providing a generic latency mechanism that can locate fetch bubbles without necessarily attributing them to a particular condition. The facility counts the events if the associated instruction reaches retirement (architecturally committed). Additionally, the user may opt to enable the PEBS facility to obtain precise information on the context of the event, e.g., EventingIP.

The supported frontend microarchitectural conditions require the following interfaces:

- The IA32_PERFEVTSELx MSR must select the FRONTEND_RETIRED event, EventSelect = C6H and UMASK = 01H.
- This event employs a new MSR, MSR_PEBS_FRONTEND, to specify the supported frontend event details, see Table 21-39.
- If precise information is desired, program the PEBS_EN_PMCx field of IA32_PEBS_ENABLE MSR as required.

Note the AnyThread field of IA32_PERFEVTSELx is ignored by the processor for the “FRONTEND_RETIRED” event.

The sub-event encodings supported by MSR_PEBS_FRONTEND.EVTSEL is given in Table 21-39.

Table 21-39. FrontEnd_Retired Sub-Event Encodings Supported by MSR_PEBS_FRONTEND.EVTSEL

Sub-Event Name	EVTSEL	Description
ANY_DSB_MISS	1H	Retired Instructions which experienced any decode stream buffer (DSB) miss.
DSB_MISS	11H	Retired Instructions which experienced a DSB miss that caused a fetch starvation cycle.
L1I_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss[1]. Additional requests to the same cache line as an in-flight L1I cache miss will not be counted.
L2_MISS	13H	The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.
ITLB_MISS	14H	The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.
STLB_MISS	15H	The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.
IDQ_READ_BUBBLES	6H	An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles. The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.

NOTES:

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR_PEBS_FRONTEND is given in Table 21-40.

Table 21-40. MSR_PEBS_FRONTEND Layout

Bit Name	Offset	Description
EVTSEL	7:0	Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 21-39.
IDQ_Bubble_Length	19:8	Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event.
IDQ_Bubble_Width	22:20	Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event.
Reserved	63:23	Reserved

The FRONTEND_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

- The event counts only retired (non-speculative) front-end events, i.e., events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a front-end (miss) event occurs outside instruction boundary (e.g., due to processor handling of architectural event), it may be reported for the next instruction to retire.

21.3.8.3 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 21.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFF-CORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 21-41.
- Supplier information (bits 29:16): see Table 21-42.
- Snoop response information (bits 37:30): see Table 21-43.

Table 21-41. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake, Kaby Lake, and Coffee Lake Microarchitectures)

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	14:3	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and uncacheable accesses.

Table 21-42 lists the supplier information field that applies to 6th generation, 7th generation and 8th generation Intel Core processors. (6th generation Intel Core processor CPUID signatures: 06_4EH and 06_5EH; 7th generation and 8th generation Intel Core processor CPUID signatures: 06_8EH and 06_9EH).

Table 21-42. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9EH)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	NO_SUPP	17	No Supplier Information available.
	L3_HITM	18	M-state initial lookup stat in L3.
	L3_HITE	19	E-state
	L3_HITS	20	S-state
	Reserved	21	Reserved
	L4_HIT	22	L4 Cache (if L4 is present in the processor).
	Reserved	25:23	Reserved
	DRAM	26	Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).

Table 21-43 lists the snoop information field that apply to processors with CPUID signatures 06_4EH, 06_5EH, 06_8EH, 06_9E, and 06_55H.

Table 21-43. MSR_OFFCORE_RSP_x Snoop Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9E, 06_55H)

Subtype	Bit Name	Offset	Description
Snoop Info	SPL_HIT	30	L4 cache super line hit (if L4 is present in the processor).
	SNOOP_NONE	31	No details on snoop-related information.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores. -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO). -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD). -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S). In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD). -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO). -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	SNOOP_NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.

21.3.8.3.1 Off-core Response Performance Monitoring for the Intel® Xeon® Scalable Processor Family

The following tables list the requestor and supplier information fields that apply to the Intel® Xeon® Scalable Processor Family.

- Transaction request type encoding (bits 15:0): see Table 21-44.
- Supplier information (bits 29:16): see Table 21-45.
- Supplier information (bits 29:16) with support for Intel® Optane™ DC Persistent Memory support: see Table 21-46.
- Snoop response information has not been changed and is the same as in (bits 37:30): see Table 21-43.

Table 21-44. MSR_OFFCORE_RSP_x Request_Type Definition (Intel® Xeon® Scalable Processor Family)

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.
DEMAND_RFO	1	Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DEMAND_CODE_RD	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
Reserved	3	Reserved.
PF_L2_DATA_RD	4	Counts the number of prefetch data reads into L2.
PF_L2_RFO	5	Counts the number of RFO Requests generated by the MLC prefetches to L2.
Reserved	6	Reserved.
PF_L3_DATA_RD	7	Counts the number of MLC data read prefetches into L3.
PF_L3_RFO	8	Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	9	Reserved.
PF_L1D_AND_SW	10	Counts data cacheline reads generated by hardware L1 data cache prefetcher or software prefetch requests.
Reserved	14:11	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

Table 21-45 lists the supplier information field that applies to the Intel Xeon Scalable Processor Family (CPUID signature: 06_55H).

Table 21-45. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_55H)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	Reserved	25:22	Reserved
	L3_MISS_LOCAL_DRAM	26	L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

Table 21-46 lists the supplier information field that applies to the Intel Xeon Scalable Processor Family (CPUID signature: 06_55H, Steppings 0x5H - 0xFH).

Table 21-46. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_55H, Steppings 0x5H - 0xFH)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	No Supplier Information available.
	L3_HIT_M	18	M-state initial lookup stat in L3.
	L3_HIT_E	19	E-state
	L3_HIT_S	20	S-state
	L3_HIT_F	21	F-state
	LOCAL_PMM	22	Local home requests that were serviced by local PMM.
	REMOTE_HOP0_PMM	23	Hop 0 Remote supplier.
	REMOTE_HOP1_PMM	24	Hop 1 Remote supplier.
	REMOTE_HOP2P_PMM	25	Hop 2 or more Remote supplier.
	L3_MISS_LOCAL_DRAM	26	L3 Miss: Local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	Hop 2 or more Remote supplier.
	Reserved	30	Reserved

21.3.8.4 Uncore Performance Monitoring Facilities on Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Cannon Lake microarchitecture introduces LLC support of up to six processor cores. To support six processor cores and eight LLC slices, existing MSRs have been rearranged and new CBo MSRs have been added. Uncore performance monitoring software drivers from prior generations of Intel Core processors will need to update the MSR addresses. The new MSRs and updated MSR addresses have been added to the Uncore PMU listing in Section 2.17.2, "MSRs Specific to 8th Generation Intel® Core™ i3 Processors," in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

21.3.9 10th Generation Intel® Core™ Processor Performance Monitoring Facility

Some 10th generation Intel® Core™ processors and some 3rd generation Intel® Xeon® Scalable Processor Family are based on Ice Lake microarchitecture. Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture. For these processors, the core PMU supports architectural performance monitoring capability with version Id 5 (see Section 21.2.5) and a host of non-architectural monitoring capabilities.

The core PMU's capability is similar to those described in Section 21.3.1 through Section 21.3.8, with some differences and enhancements summarized in Table 21-47.

Table 21-47. Core PMU Summary of the Ice Lake Microarchitecture

Box	Ice Lake Microarchitecture	Skylake, Kaby Lake and Coffee Lake Microarchitectures	Comment
Architectural PerfMon version	5	4	See Section 21.2.5.
Number of programmable counters per thread	8	4	Use CPUID to determine number of counters. See Section 21.2.1.
PEBS: Basic functionality	Yes	Yes	See Section 21.3.9.1.
PEBS record format encoding	0100b	0011b	See Section 21.6.2.4.2.
Extended PEBS	PEBS is extended to all Fixed and General Purpose counters and to all performance monitoring events.	No	See Section 21.9.1.
Adaptive PEBS	Yes	No	See Section 21.9.2.
Performance Metrics	Yes (4)	No	See Section 21.3.9.3.
PEBS-PDIR	IA32_FIXED0 only (Corresponding counter control MSRs must be enabled.)	IA32_PMC1 only.	

21.3.9.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 10th generation Intel Core processors provides a number of enhancements relative to PEBS in processors based on the Skylake, Kaby Lake, and Coffee Lake microarchitectures. Enhancement of the PEBS facility with Extended PEBS and Adaptive PEBS features is described in detail in Section 21.9.

The 3rd generation Intel Xeon Scalable Family of processors based on the Ice Lake microarchitecture introduce EPT-friendly PEBS. This allows EPT violations and other VM Exits to be taken on PEBS accesses to the DS Area. See Section 21.9.5 for details.

21.3.9.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 21.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFF-CORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-[N1].
- Response type encoding (bits 16-37) of
  - — Supplier information: see Table [18-N2].
  - — Snoop response information: see Table [18-N3].
- All transactions are tracked at cacheline granularity except some in request type OTHER.

Table 21-48. MSR_OFFCORE_RSP_x Request_Type Definition (Processors Based on Ice Lake Microarchitecture)

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data and page table entry reads.
DEMAND_RFO	1	Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.
DEMAND_CODE_RD	2	Counts demand instruction fetches and instruction prefetches targeting the L1 instruction cache.
Reserved	3	Reserved

Table 21-48. MSR_OFFCORE_RSP_x Request_Type Definition (Processors Based on Ice Lake Microarchitecture)

Bit Name	Offset	Description
HWPF_L2_DATA_RD	4	Counts hardware generated data read prefetches targeting the L2 cache.
HWPF_L2_RFO	5	Counts hardware generated prefetches for exclusive ownership (RFO) targeting the L2 cache.
Reserved	6	Reserved
HWPF_L3	9:7 and 13[1]	Counts hardware generated prefetches of any type targeting the L3 cache.
HWPF_L1D_AND_SWPF	10	Counts hardware generated data read prefetches targeting the L1 data cache and the following software prefetches (PREFETCHNTA, PREFETCHT0/1/2).
STREAMING_WR	11	Counts streaming stores.
Reserved	12	Reserved
Reserved	14	Reserved
OTHER	15	Counts miscellaneous requests, such as I/O and un-cacheable accesses.

NOTES:

1. All bits need to be set to 1 to count this type.

Ice Lake microarchitecture has added a new category of Response subtype, called a Combined Response Info. To count a feature in this type, all the bits specified must be set to 1.

A valid response type must be a non-zero value of the following expression:

Any | ['OR' of Combined Response Info Bits | [('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]]

If “ANY” bit[16] is set, other response type bits [17-39] are ignored.

Table 21-49 lists the supplier information field that applies to processors based on Ice Lake microarchitecture.

Table 21-49. MSR_OFFCORE_RSP_x Supplier Info Field Definition (Processors Based on Ice Lake Microarchitecture)

Subtype	Bit Name	Offset	Description
Common	Any	16	Catch all value for any response types.
Combined Response Info	DRAM	26, 31, 32[1]	Requests that are satisfied by DRAM.
	NON_DRAM	26, 37[1]	Requests that are satisfied by a NON_DRAM system component. This includes MMIO transactions.
	L3_MISS	22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37[1]	Requests that were not supplied by the L3 Cache. The event includes some currently reserved bits in anticipation of future memory designs.
Supplier Info	L3_HIT	18,19, 20[1]	Requests that hit in L3 cache. Depending on the snoop response the L3 cache may have retrieved the cacheline from another core’s cache.
Reserved		17, 21:25, 27:29	Reserved.

NOTES:

1. All bits need to be set to 1 to count this type.

Table 21-50 lists the snoop information field that applies to processors based on Ice Lake microarchitecture.

Table 21-50. MSR_OFFCORE_RSP_x Snoop Info Field Definition
(Processors Based on Ice Lake Microarchitecture)

Subtype	Bit Name	Offset	Description
Snoop Info	Reserved	30	Reserved.
	SNOOP_NOT_NEEDED	32	No snoop was needed to satisfy the request.
	SNOOP_MISS	33	A snoop was sent and none of the snooped caches contained the cacheline.
	SNOOP_HIT_NO_FWD	34	A snoop was sent and hit in at least one snooped cache. The unmodified cacheline was not forwarded back, because the L3 already has a valid copy.
	Reserved	35	Reserved.
	SNOOP_HITM	36	A snoop was sent and the cacheline was found modified in another core's caches. The modified cacheline was forwarded to the requesting core.

21.3.9.3 Performance Metrics

The Ice Lake core PMU provides built-in support for Top-down Microarchitecture Analysis (TMA) method level 1 metrics. These metrics are always available to cross-validate performance observations, freeing general purpose counters to count other events in high counter utilization scenarios. For more details about the method, refer to Top-Down Analysis Method chapter (Appendix B.1) of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

A new MSR called MSR_PERF_METRICS reports the metrics directly. Software can check (and/or expose to its guests) the availability of the PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15). For additional details on this MSR, refer to Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

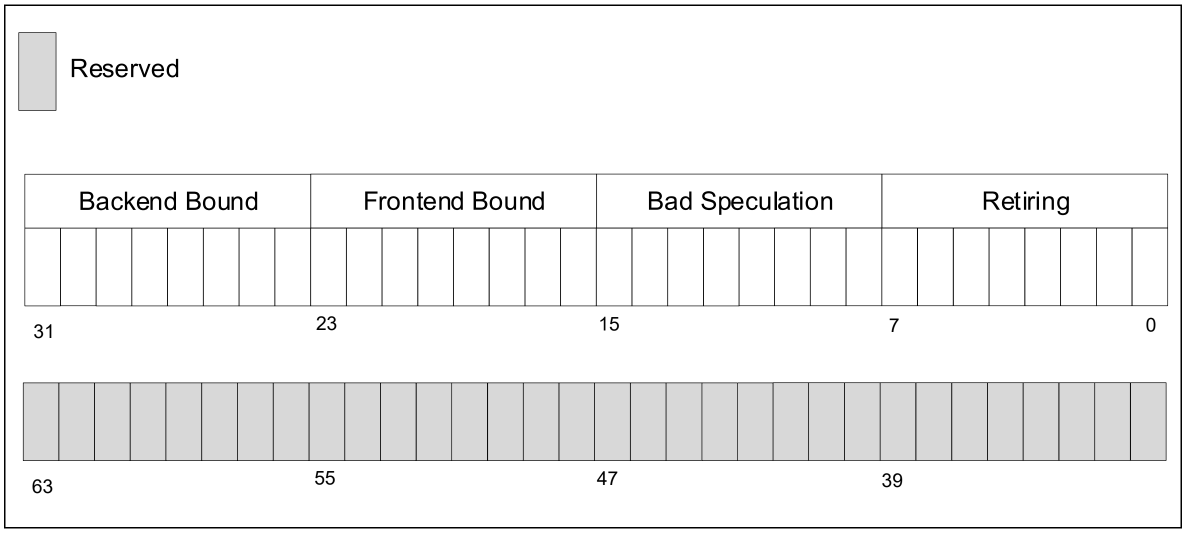


Figure 21-39. MSR_PERF_METRICS Definition

This register exposes the four TMA Level 1 metrics. The lower 32 bits are divided into four 8-bit fields, as shown by the above figure, each of which is an integer fraction of 255.

To support built-in performance metrics, new bits have been added to the following MSRs:

- IA32_PERF_GLOBAL_CTRL. EN_PERF_METRICS[48]: If this bit is set and fixed-function performance-monitoring counter 3 is enabled, built-in performance metrics are enabled.
- IA32_PERF_GLOBAL_STATUS_SET. SET_OVF_PERF_METRICS[48]: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.
- IA32_PERF_GLOBAL_STATUS_RESET. RESET_OVF_PERF_METRICS[48]: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.
- IA32_PERF_GLOBAL_STATUS. OVF_PERF_METRICS[48]: If this bit is set, it indicates that a PERF_METRICS-related resource has overflowed and a PMI is triggered[1]. If this bit is clear, no such overflow has occurred.

NOTE

Software has to synchronize, e.g., re-start, fixed-function performance-monitoring counter 3 as well as PERF_METRICS when either bit 35 or 48 in IA32_PERF_GLOBAL_STATUS is set. Otherwise, PERF_METRICS may return undefined values.

The values in MSR_PERF_METRICS are derived from fixed-function performance-monitoring counter 3. Software should start both registers, PERF_METRICS and fixed-function performance-monitoring counter 3, from zero. Additionally, software is recommended to periodically clear both registers in order to maintain accurate measurements for certain scenarios that involve sampling metrics at high rates.

In order to save/restore PERF_METRICS, software should follow these guidelines:

- PERF_METRICS and fixed-function performance-monitoring counter 3 should be saved and restored together.
- To ensure that PERF_METRICS and fixed-function performance-monitoring counter 3 remain synchronized, both should be disabled during both save and restore. Software should enable/disable them atomically, with a single write to IA32_PERF_GLOBAL_CTRL to set/clear both EN_PERF_METRICS[bit 48] and EN_FIXED_CTR3[bit 35].
- On state restore, fixed-function performance-monitoring counter 3 must be restored **before** PERF_METRICS, otherwise undefined results may be observed.

21.3.10 12th and 13th Generation Intel® Core™ Processors, and 4th and 5th Generation Intel® Xeon® Scalable Processor Family Performance Monitoring Facility

The 12th generation Intel® Core™ processor supports Alder Lake performance hybrid architecture. These processors offer a unique combination of Performance and Efficient-cores (P-core and E-core). The P-core is based on Golden Cove microarchitecture and the E-core is based on Gracemont microarchitecture. The 13th generation Intel® Core™ processor supports Raptor Lake performance hybrid architecture, utilizing both Raptor Cove cores and enhanced Gracemont cores. The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture utilizing Golden Cove cores. The 5th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture utilizing Raptor Cove cores. These processors all report architectural performance monitoring version ID = 5 and support non-architectural monitoring capabilities described in this section.

21.3.10.1 P-core Performance Monitoring Unit

The P-core PMU's capability is similar to those described in Section 21.3.1 through Section 21.3.9, with some differences and enhancements summarized in Table 21-51.

1. An overflow of fixed-function performance-monitoring counter 3 should normally happen first if software follows Intel's recommendations.

Table 21-51. Core PMU Summary of the Golden Cove Microarchitecture

Box	Golden Cove Microarchitecture	Ice Lake Microarchitecture	Comment
Architectural PerfMon version	5	5	See Section 21.2.5.
Event-Counter Restrictions	Simplified identification		Counters 4-7 support a subset of events. See Section 21.3.10.1.2.
Performance Metrics	Yes (12)	Yes (4)	See Section 21.3.9.3.
PEBS: Baseline, record format	Yes 0100b	Yes 0100b	See Section 21.3.9.
PEBS: EPT-friendly	Yes	No; debuts in Ice Lake server microarchitecture	See Section 21.6.2.4.2.
PEBS: Precise Distribution	IA32_FIXED0 instruction-granularity PDist on IA32_PMC0	IA32_FIXED0 cycle-granularity No PDist	See Section 21.9.6.
PEBS: Load Latency	Instruction latency Cache latency Access info fields (5)	Instruction latency Access info fields (3)	See Section 21.9.7.
PEBS: Store Latency	Cache latency Access info fields (3)	None	See Section 21.9.8.
PEBS: Intel TSX support	Abort info fields (9)	Abort info fields (8)	See Section 21.3.6.5.1. (Intel Xeon processor only feature.)

21.3.10.1.1 P-core Perf Metrics Extensions

For 12th generation Intel Core processor P-cores, the core PMU supports the built-in metrics that were introduced in the Ice Lake microarchitecture PMU. This core PMU extends the PERF_METRICS MSR to feature TMA method level 2 metrics, as shown in Figure 21-40.

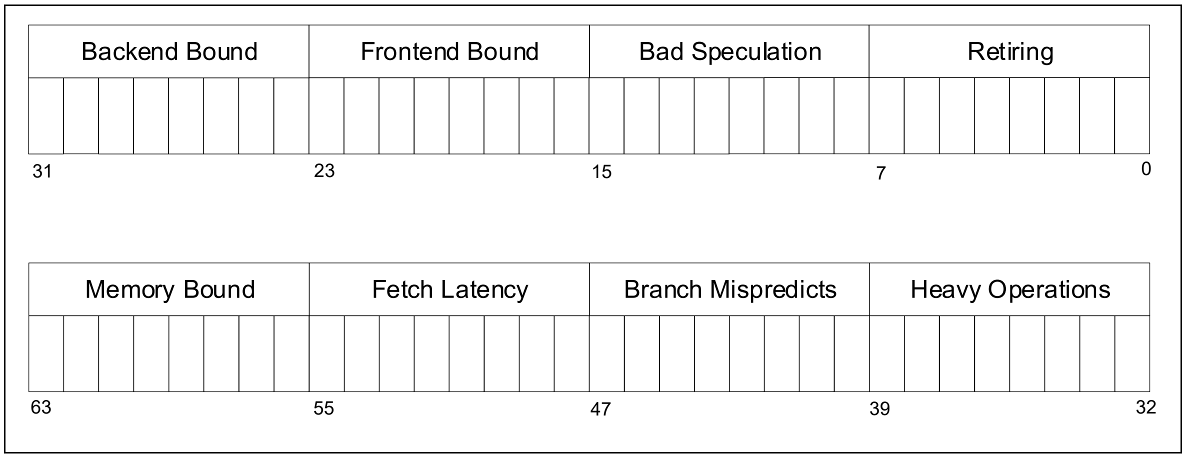


Figure 21-40. PERF_METRICS MSR Definition for 12th Generation Intel® Core™ Processor P-core

The lower half of the register is the TMA level 1 metrics (legacy). The upper half is also divided into four 8-bit fields, each of which is an integer fraction of 255. Additionally, each of the new level 2 metrics in the upper half is a subset of the corresponding level 1 metric in the lower half (that is, its parent node per the TMA hierarchy). This enables software to deduce the other four level 2 metrics by subtracting corresponding metrics as shown in Figure 21-41.

```
Light_Operations = Retiring - Heavy_Operations
Machine_Clears = Bad_Speculation - Branch_Mispredicts
Fetch_Bandwidth = Frontend_Bound - Fetch_Latency
Core_Bound = Backend_Bound - Memory_Bound
```

Figure 21-41. Deducing Implied Level 2 Metrics in the Core PMU for12th Generation Intel® Core™ Processor P-core

The PERF_METRICS MSR and fixed-function performance-monitoring counter 3 of the core PMU feature 12 metrics in total that cover all level 1 and level 2 nodes of the TMA hierarchy.

21.3.10.1.2 P-core Counter Restrictions Simplification

The 12th generation Intel Core processor P-core allows identification of performance monitoring events with counter restrictions based on event encodings. The general rule is: Event Codes < 0x90 are restricted to general-purpose performance-monitoring counters 0-3. Event Codes ≥ 0x90 are likely to have no restrictions. Table 21-52 lists the exceptions to this rule.

Table 21-52. Special Performance Monitoring Events with Counter Restrictions

Event Encoding[1]	Event Name	Counter Restriction
xx3C	CPU_CLK_UNHALTED.*	0-7 (No restriction for all architectural events.)
xx2E	LONGEST_LAT_CACHE.*	
xxDx	MEM_*_RETIRED.*	0-3
01A3, 02A3, 08A3	Some CYCLE_ACTIVITY sub-events	0-3
02CD	MEM_TRANS_RETIRED.STORE_SAMPLE	0
04A4	TOPDOWN.BAD_SPEC_SLOTS	0
08A4	TOPDOWN.BR_MISPREDICT_SLOTS	
xxCE	AMX_OPS_RETIRED	0

NOTES:

1. Linux perf rUUEE syntax, where UU is the Unit Mask field and EE is the Event Select (also known as Event Code) field in the IA32_PERFEVTSELx MSRs.

21.3.10.1.3 P-core Off-core Response Facility

For the 12th generation Intel Core processor P-core, the Off-core Response (OCR) Facility is similar to that described in Section 21.3.9.2.

The following enhancements are introduced for the Request_Type of MSR_OFFCORE_RSP_x:

- WB (bits 3 and 12): Count writeback (modified or non-modified) transactions by core caches.
- HWPF_L1D (bit 10): Counts hardware generated data read prefetches targeting the L1 data cache (only).
- SWPF_READ (bit 14): Counts software generated data read prefetches by the PREFETCHNTA and PREFETCHT0/1/2 instructions.

21.3.10.2 E-core Performance Monitoring Unit

The core PMU capabilities on the 12th generation Intel Core processor E-core are summarized in Table 21-53 below.

Table 21-53. Core PMU Summary of the Gracemont Microarchitecture

Box	Gracemont Microarchitecture	Tremont Microarchitecture	Comment
Number of fixed-function performance-monitoring counters per core	3	3	Use CPUID to enumerate number of counters. See Section 21.2.1.
Number of general-purpose counters per core	6	4	Use CPUID to enumerate number of counters. See Section 21.2.1.
Architectural Performance Monitoring version ID	5	5	See Section 21.2.5.
PEBS record format encoding	0100b	0100b	See Section 21.5.5.
EPT-friendly PEBS support	Yes	No	See Section 21.9.5.
Extended PEBS	Yes	Yes	See Section 21.9.1.
Adaptive PEBS	Yes	Yes	See Section 21.9.2.
Precise distribution (PDist) PEBS	IA32_PMC0 and IA32_FIXED_CTR0	IA32_PMC0 and IA32_FIXED_CTR0	PDist eliminates skid, see Section 21.9.3, Section 21.9.4, and Section 21.9.6.
PEBS Latency	Load and Store Latency	No	See Section 21.3.10.2.1, Section 21.3.10.2.2, Section 21.9.7, and Section 21.9.8.
PEBS Output	DS Save Area or Intel® Processor Trace	DS Save Area or Intel® Processor Trace	See Section 21.5.5.2.1.
Offcore Response	MSR 01A6H and 01A7H, each core has its own register, extended request and response types.	MSR 1A6H and 1A7H, each core has its own register, extended request and response types.	See Section 21.5.5.4.

21.3.10.2.1 E-core PEBS Load Latency

The 12th generation Intel Core processor E-core includes PEBS Load Latency support similar to that described in Section 21.9.7.

When a programmable counter is configured to count MEM_UOPS_RETIRED.LOAD_LATENCY_ABOVE_THRESHOLD (IA32_PERFEVTSELx[15:0] = 0xD005, with CMASK=0 and INV=0), selected load operations whose latency exceeds the threshold provided in MSR_PEBS_LD_LAT_THRESHOLD (MSR 03F6H) will be counted. If a PEBS record is generated on overflow of this counter, the Memory Access Latency and Memory Auxiliary Info data is reported in the Memory Access Info group (Section 21.9.2.2.2). The formats of these fields are shown in Table 21-54 and Table 21-98.

Table 21-54. E-core PEBS Memory Access Info Encoding

Bit(s)	Field	Description
3:0	Data Source	The source of the data; see Table 21-55.
4	Lock	0: The operation was not part of a locked transaction. 1: The operation was part of a locked transaction.
5	STLB_MISS	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.

Table 21-54. E-core PEBS Memory Access Info Encoding (Contd.)

Bit(s)	Field	Description
6	ST_FWD_BLK	0: Load did not get a store forward block. 1: Load got a store forward block.
63:7	Reserved	Reserved

For details on E-core PEBS memory access latency encoding, see the Access Latency Field in Table 21-98.

Table 21-55. E-core PEBS Data Source Encodings

Encoding	Description
00H	Unknown Data Source (the processor could not retrieve the origin of this request) and MMIO. Memory mapped I/O hit.
01H	L1 HIT. This request was satisfied by the L1 data cache. (Minimal latency core cache hit.)
02H	FB HIT. Outstanding core cache miss to same cache-line address was already underway. (Pending core cache hit.)
03H	L2 HIT. This request was satisfied by the L2 cache.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HITE. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found (clean).
06H	L3 HITM. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where a modified copy was found.
07H	Reserved.
08H	L3 HITF. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where a shared or forwarding copy was found.
09H	Reserved.
0AH	L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).
0BH	Reserved.
0CH	Reserved.
0DH	Reserved.
0EH	I/O. Request of input/output operation.
0FH	The request was to uncacheable memory.

21.3.10.2.2 E-core PEBS Store Latency

The 12th generation Intel Core processor E-core includes PEBS Store Latency support. When a programmable counter is configured to count MEM_UOPS_RETIRED.STORE_LATENCY (IA32_PERFEVTSELx[15:0] = 0xD006, with CMASK=0 and INV=0), all store operations will be counted. If a PEBS record is generated on overflow of this counter, the Memory Access Latency and Memory Auxiliary Info data is reported in the Memory Access Info group (Section 18.9.2.2.2). The formats of these fields are shown in Table 21-54 and Table 21-98.

21.3.10.2.3 E-core Precise Distribution (PDist) Support

The 12th generation Intel Core processor E-core supports PEBS with Precise Distribution (PDist) on IA32_PMC0 and IA32_FIXED_CTR0. All precise events support PDist save for UOPS_RETIRED. See Section 21.9.6 for additional details on PDist.

21.3.10.2.4 E-core Enhanced Off-core Response

Event number 0B7H support off-core response monitoring using an associated configuration MSR, MSR_OFF-CORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. There are unique pairs of MSR_OFFCORE_RSPx registers per core. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as follows:

- Bits 15:0 and bits 49:44 specify the request type of a transaction request to the uncore. This is described in Table 21-56.
- Bits 30:16 specify Response Type information or an L2 Hit, and is described in Table 21-79.
- If L2 misses, then bits 37:31 can be used to specify snoop response information and is described in Table 21-80.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 21.5.2.3 for details.

Table 21-56. MSR_OFFCORE_RSPx Request Type Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data reads.
DEMAND_RFO	1	Counts all demand reads for ownership (RFO) requests and software based prefteches for exclusive ownership (prefetchw).
DEMAND_CODE_RD	2	Counts demand instruction fetches and L1 instruction cache prefetches.
COREWB_M	3	Counts modified write backs from L1 and L2.
HWPF_L2_DATA_RD	4	Counts prefetch (that bring data to L2) data reads.
HWPF_L2_RFO	5	Counts all prefetch (that bring data to L2) RFOs.
HWPF_L2_CODE_RD	6	Counts all prefetch (that bring data to MLC only) code reads.
HWPF_L3_DATA_RD	7	Counts L3 cache hardware prefetch data reads (written to the L3 cache only).
HWPF_L3_RFO	8	Counts L3 cache hardware prefetch RFOs (written to the L3 cache only) .
HWPF_L3_CODE_RD	9	Counts L3 cache hardware prefetch code reads (written to the L3 cache only).
HWPF_L1D_AND_SWPF	10	Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).
STREAMING_WR	11	Counts all streaming stores.
COREWB_NONM	12	Counts non-modified write backs from L2.
RSVD	14:13	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O accesses that have any response type.
UC_RD	44	Counts uncached memory reads (PRd, UCRdF).
UC_WR	45	Counts uncached memory writes (WiL).
PARTIAL_STREAMING_WR	46	Counts partial (less than 64 byte) streaming stores (WCiL).
FULL_STREAMING_WR	47	Counts full, 64 byte streaming stores (WCiLF).
L1WB_M	48	Counts modified WriteBacks from L1 that miss the L2.
L2WB_M	49	Counts modified WriteBacks from L2.

21.3.10.3 Unhalted Reference Cycles

The Unhalted Reference Cycles architectural performance monitoring event is enhanced to count at TSC-rate in the 12th generation Intel Core processor P-core when used on a general-purpose PMC. This enhancement makes it consistent with the fixed-function counter 2 and the E-core. As a result, this event is kept enumerated in CPUID leaf 0AH.EBX (unlike prior hybrid parts).

21.3.11 Intel® Series 2 Core™ Ultra Processor Performance Monitoring Facility

The Intel® Series 2 Core™ Ultra processor supports Lunar Lake performance hybrid architecture. This processor offers a combination of Performance and Efficient-cores (P-core and E-core). The P-core is based on Lion Cove microarchitecture and the E-core is based on Skymont microarchitecture. This processor reports architectural performance monitoring version ID = 6 and supports non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 6 capabilities are described in Section 21.2.6.

21.3.11.1 P-core Performance Monitoring Unit

The core PMU capabilities on the Intel Series 2 Core Ultra processor P-core are similar to those described in Section 21.3.1 through Section 21.3.10, with some differences and enhancements summarized in Table 21-57.

Table 21-57. Core PMU Summary of the Lion Cove Microarchitecture

Box	Lion Cove Microarchitecture	Golden Cove and Redwood Cove Microarchitectures	Comment
Architectural Performance Monitoring version ID	6	5	See Section 21.2.6.
Number of general-purpose counters per core	10	8	Use CPUID to enumerate number of counters. See Section 21.2.1 and Section 21.2.9.
Number of architectural performance-monitoring events	12	8 (Golden Cove) 11 (Redwood Cove)	See Section 21.2.7.
Event-Counter Restrictions	Mostly homogeneous general counters.	Simplified identification of events supported on counters 4-7.	Few counter restrictions may apply; see Section 21.3.1.1.1.
Performance Metrics	12 metrics Metrics clear mode or read-only.	12 metrics Read-only.	See the RDPMC instruction in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.
OCR: MSR_OFFCORE_RSP_0/1	0FFF FFFF FFFFH	003F FFFF FFFFH	See Section 2.17.9 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.
PEBS: Baseline	Yes	Yes	See Section 21.8.
PEBS record format encoding	0110b	0101b	See Section 21.9.2.2.
PEBS: Precise Distribution	IA32_FIXED0 instruction-granularity. PDist on IA32_PMC0 and IA32_PMC1.	IA32_FIXED0 instruction-granularity. PDist on IA32_PMC0.	See Section 21.9.6.
PEBS: Data Source field	5-bits	4-bits	See Section 21.9.7.
LBR: Event Logging	Yes	No	See Section 20.1.3.6.
Intel PT: TNT Disable	Yes	No	See Chapter 34 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C.

21.3.11.1.1 P-core Homogeneous General Counters

The Lion Cove PMU enhances general-counters to support most of the performance monitoring events. The remaining events that do have counter restrictions are summarized in next Table 21-58.

Table 21-58. Performance Monitoring Events with Counter Restrictions in Lion Cove PMU

Event Encoding[1]	Event Name	Counter Restriction
xx20	OFFCORE_REQUESTS_OUTSTANDING.*	0-3
0148	L1D_PENDING.*	2
0175	INST_DECODED.DECODERS	2
08A3, 0CA3	CYCLE_ACTIVITY.*_L1D_MISS	2
04A4, 08A4, 10A4	TOPDOWN.BAD_SPEC_SLOTS, TOPDOWN.BR_MISPREDICT_SLOTS, TOPDOWN.MEMORY_BOUND_SLOTS	0
01B1	UOPS_EXECUTED.*	3
xxDx	MEM_INST_RETIRED.*, MEM_LOAD*_RETIRED.*	0-3
02CD	MEM_TRANS_RETIRED.STORE_SAMPLE	0-1

NOTES:

1. Linux perf rUUEE syntax, where UU is the Unit Mask field and EE is the Event Select (also known as Event Code) field in the IA32_PERFEVTSELx MSRs.

21.3.11.2 E-core Performance Monitoring Unit

Skymont microarchitecture performance monitoring capabilities are similar to Crestmont microarchitecture capabilities, with the following extensions:

- Support for fixed counters 4, 5, and 6 (see Section 21.2.9.4)
- Architecturally defined events: TMA L1 and LBR Inserts (see Section 21.2.9.6 and Section 20.1.3.6)
- PEBS Counter Snapshotting (see Section 21.9.10)
- Auto Counter Reload (see Section 21.10)

The core PMU capabilities on the Intel Series 2 Core Ultra processor E-core are summarized in Table 21-59.

Table 21-59. Core PMU Summary of the Skymont Microarchitecture

Box	Skymont Microarchitecture	Crestmont Microarchitecture	Comment
Architectural Performance Monitoring version ID	6	5	See Section 21.2.6.
Number of fixed-function performance-monitoring counters per core	6 (0, 1, 2, 4, 5, 6)	3 (0, 1, 2)	Use CPUID to enumerate number of counters. See Section 21.2.1 and Section 21.2.9.4.
Number of general-purpose counters per core	8	8	Use CPUID to enumerate number of counters. See Section 21.2.1.
PEBS record format encoding	0110b	0101b	See Section 21.3.10.
Auto Counter Reload (ACR)	Yes	No	See Section 21.10.

21.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS)

NOTE

This section also applies to the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture.

21.4.1 Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.

A complete description of the tile and untile PMU programming interfaces for Intel Xeon Phi processors based on the Knights Landing microarchitecture can be found in the Technical Document section at http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

A tile contains a pair of cores attached to a shared L2 cache and is similar to those found in Intel Atom® processors based on the Silvermont microarchitecture. The processor provides several new capabilities on top of the Silvermont performance monitoring facilities.

The processor supports architectural performance monitoring capability with version ID 3 (see Section 21.2.3) and a host of non-architectural performance monitoring capabilities. The processor provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in the processor also uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 21-6 and described in Section and Section 21.2.3. The processor supports AnyThread counting in three architectural performance monitoring events.

21.4.1.1 Enhancements of Performance Monitoring in the Intel® Xeon Phi™ Processor Tile

The Intel® Xeon Phi™ processor tile includes the following enhancements to the Silvermont microarchitecture.

- AnyThread support. This facility is limited to following three architectural events: Instructions Retired, Unhalted Core Cycles, Unhalted Reference Cycles using IA32_FIXED_CTR0-2 and Unhalted Core Cycles, Unhalted Reference Cycles using IA32_PERFEVTSELx.
- PEBS-DLA (Processor Event-Based Sampling-Data Linear Address) fields. The processor provides memory address in addition to the Silvermont PEBS record support on select events. The PEBS recording format as reported by IA32_PERF_CAPABILITIES [11:8] is 2.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor tile to subsystems outside the tile (untile). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx. Two cores do not share the off-core response MSRs. Knights Landing expands off-core response capability to match the processor untile changes.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests. This facility is updated to match the processor untile changes.

21.4.1.1.1 Processor Event-Based Sampling

The processor supports processor event based sampling (PEBS). PEBS is supported using IA32_PMC0 (see also Section 19.4.9, “BTS and DS Save Area”).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 21.6.2.4).

The list of PEBS events supported in the processor is shown in the following table.

Table 21-60. PEBS Performance Events for Knights Landing Microarchitecture

Event Name	Event Select	Sub-event	UMask	Data Linear Address Support
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		CALL	F9H	No
		REL_CALL	FDH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		FAR_BRANCH	BFH	No
		RETURN	F7H	No
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		RETURN	F7H	No
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H	Yes
		L2_MISS_LOADS	04H	Yes
		DLTB_MISS_LOADS	08H	Yes
RECYCLEQ	03H	LD_BLOCK_ST_FORWARD	01H	Yes
		LD_SPLITS	08H	Yes

The PEBS record format 2 supported by processors based on the Knights Landing microarchitecture is shown in Table 21-61, and each field in the PEBS record is 64 bits long.

Table 21-61. PEBS Record Format for Knights Landing Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	PSDLA
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

21.4.1.1.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFF-CORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 21-62 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 21-62. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

Some of the MSR_OFFCORE_RESP [0,1] register bits are not valid in this processor and their use is reserved. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 registers are defined in Table 21-63. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 21.5.2.3 for details.

Table 21-63. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers

Main	Sub-field	Bit	Name	Description
Request Type		0	DEMAND_DATA_RD	Demand cacheable data and L1 prefetch data reads.
		1	DEMAND_RFO	Demand cacheable data writes.
		2	DEMAND_CODE_RD	Demand code reads and prefetch code reads.
		3	Reserved	Reserved.
		4	Reserved	Reserved.
		5	PF_L2_RFO	L2 data RFO prefetches (includes PREFETCHW instruction).
		6	PF_L2_CODE_RD	L2 code HW prefetches.
		7	PARTIAL_READS	Partial reads (UC or WC).
		8	PARTIAL_WRITES	Partial writes (UC or WT or WP). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		9	UC_CODE_READS	UC code reads.
		10	BUS_LOCKS	Bus locks and split lock requests.
		11	FULL_STREAMING_STORES	Full streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		12	SW_PREFETCH	Software prefetches.
		13	PF_L1_DATA_RD	L1 data HW prefetches.
		14	PARTIAL_STREAMING_STORES	Partial streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		15	ANY_REQUEST	Account for any requests.
Response Type	Any	16	ANY_RESPONSE	Account for any response.
	Data Supply from Untile	17	NO_SUPP	No Supplier Details.
		18	Reserved	Reserved.

Table 21-63. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers (Contd.)

Main	Sub-field	Bit	Name	Description
		19	L2_HIT_OTHER_TILE_NEAR	Other tile L2 hit E Near.
		20	Reserved	Reserved.
		21	MCDRAM_NEAR	MCDRAM Local.
		22	MCDRAM_FAR_OR_L2_HIT_OTHER_TILE_FAR	MCDRAM Far or Other tile L2 hit far.
		23	DRAM_NEAR	DRAM Local.
		24	DRAM_FAR	DRAM Far.
	Data Supply from within same tile	25	L2_HITM_THIS_TILE	M-state.
		26	L2_HITE_THIS_TILE	E-state.
		27	L2_HITS_THIS_TILE	S-state.
		28	L2_HITF_THIS_TILE	F-state.
		29	Reserved	Reserved.
		30	Reserved	Reserved.
	Snoop Info; Only Valid in case of Data Supply from Untile	31	SNOOP_NONE	None of the cores were snooped.
		32	NO_SNOOP_NEEDED	No snoop was needed to satisfy the request.
		33	Reserved	Reserved.
		34	Reserved	Reserved.
		35	HIT_OTHER_TILE_FWD	Snoop request hit in the other tile with data forwarded.
		36	HITM_OTHER_TILE	A snoop was needed and it HitM-ed in other core's L1 cache. HitM denotes a cache-line was in modified state before effect as a result of snoop.
		37	NON_DRAM	Target was non-DRAM system address. This includes MMIO transactions.
Outstanding requests	Weighted cycles	38	OUTSTANDING (Valid only for MSR_OFFCORE_RESP0. Should only be used on PMC0. This bit is reserved for MSR_OFFCORE_RESP1).	If set, counts total number of weighted cycles of any outstanding offcore requests with data response. Valid only for OFFCORE_RESP_0 event. Should only be used on PMC0. This bit is reserved for OFFCORE_RESP_1 event.

21.4.1.1.3 Average Offcore Request Latency Measurement

Measurement of average latency of offcore transaction requests can be enabled using MSR_OFFCORE_RSP0.[bit 38] with the choice of request type specified in MSR_OFFCORE_RSP0.[bit 15:0].

Refer to Section 21.5.2.3, "Average Offcore Request Latency Measurement," for typical usage. Note that MSR_OFFCORE_RESPx registers are not shared between cores in Knights Landing. This allows one core to measure average latency while other core is measuring different offcore response events.

21.5 PERFORMANCE MONITORING (INTEL ATOM® PROCESSORS)

21.5.1 Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 21.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_-FIXED_CTR1, IA32_FIXED_CTR2).

NOTE

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 21.2.1.

Non-architectural performance monitoring in Intel Atom processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

Architectural and non-architectural performance monitoring events in 45 nm and 32 nm Intel Atom processors support thread qualification using bit 21 (AnyThread) of IA32_PERFEVTSELx MSR, i.e., if IA32_PERFEVT-SELx.AnyThread =1, event counts include monitored conditions due to either logical processors in the same processor core.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 21-6 and described in Section and Section 21.2.3.

Valid event mask (Umask) bits can be found at: https://perfmon-events.intel.com/. The UMASK field may contain sub-fields that provide the same qualifying actions like those listed in Table 21-81, Table 21-82, Table 21-83, and Table 21-84. One or more of these sub-fields may apply to specific events on an event-by-event basis. Precise Event Based Monitoring is supported using IA32_PMC0 (see also Section 19.4.9, “BTS and DS Save Area”).

21.5.2 Performance Monitoring for Silvermont Microarchitecture

Intel processors based on the Silvermont microarchitecture report architectural performance monitoring versionID = 3 (see Section 21.2.3) and a host of non-architectural monitoring capabilities. Intel processors based on the Silvermont microarchitecture provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2). Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Non-architectural performance monitoring in the Silvermont microarchitecture uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 21-6 and described in Section and Section 21.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32_PERFEVTSELx MSR.

21.5.2.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- The width of counter reported by CPUID.0AH:EAX[23:16] is 40 bits.

- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

21.5.2.1.1 Processor Event Based Sampling (PEBS)

In the Silvermont microarchitecture, the PEBS facility can be used with precise events. PEBS is supported using IA32_PMC0 (see also Section 19.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 21.6.2.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 21-64.

Table 21-64. PEBS Performance Events for the Silvermont Microarchitecture

Event Name	Event Select	Sub-event	UMask
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H
		L2_MISS_LOADS	04H
		DLTB_MISS_LOADS	08H
		HITM	20H
REHABQ	03H	LD_BLOCK_ST_FORWARD	01H
		LD_SPLITS	08H

PEBS Record Format The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 21-65, and each field in the PEBS record is 64 bits long.

Table 21-65. PEBS Record Format for the Silvermont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Reserved
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

21.5.2.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFF-CORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 21-66 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

In the Silvermont microarchitecture, each MSR_OFFCORE_RSPx is shared by two processor cores.

Table 21-66. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are shown in Figure 21-42 and Figure 21-43. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 21.5.2.3 for details.

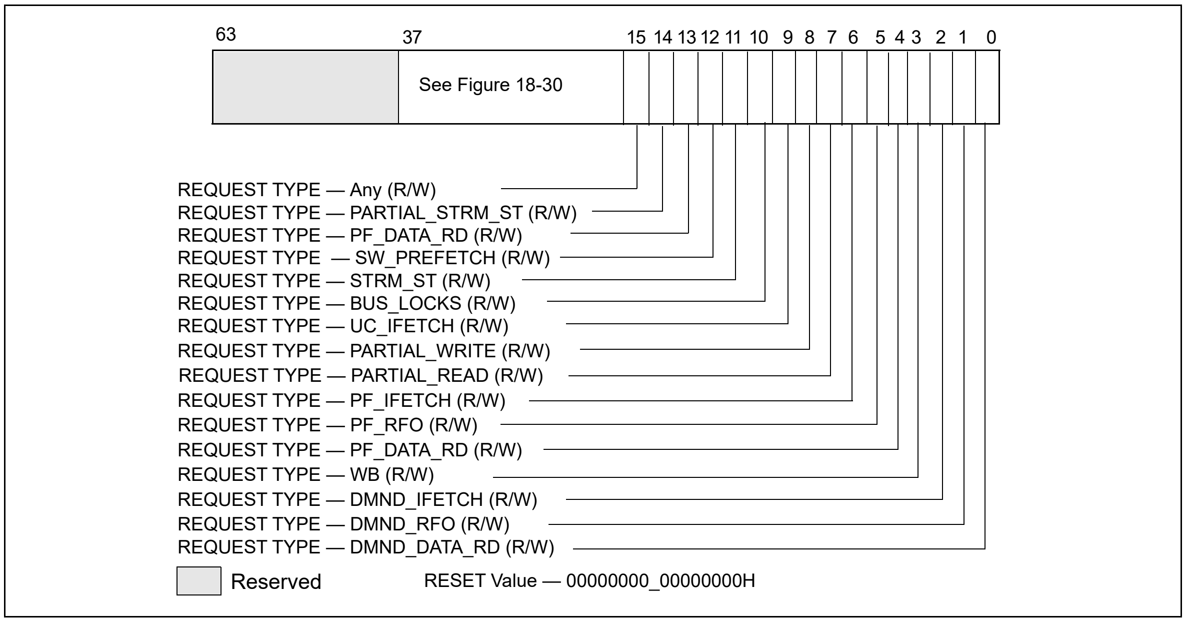

63
37
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
See Figure 18-30
REQUEST TYPE — Any (R/W)
REQUEST TYPE — PARTIAL_STRM_ST (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — SW_PREFETCH (R/W)
REQUEST TYPE — STRM_ST (R/W)
REQUEST TYPE — BUS_LOCKS (R/W)
REQUEST TYPE — UC_IFETCH (R/W)
REQUEST TYPE — PARTIAL_WRITE (R/W)
REQUEST TYPE — PARTIAL_READ (R/W)
REQUEST TYPE — PF_IFETCH (R/W)
REQUEST TYPE — PF_RFO (R/W)
REQUEST TYPE — PF_DATA_RD (R/W)
REQUEST TYPE — WB (R/W)
REQUEST TYPE — DMND_IFETCH (R/W)
REQUEST TYPE — DMND_RFO (R/W)
REQUEST TYPE — DMND_DATA_RD (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 21-42. Request_Type Fields for MSR_OFFCORE_RSPx

Table 21-67. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.
WB	3	Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT, and WP).
UC_IFETCH	9	Counts the number of UC instruction fetches.
BUS_LOCKS	10	Bus lock and split lock requests.
STRM_ST	11	Streaming store requests.
SW_PREFETCH	12	Counts software prefetch requests.
PF_DATA_RD	13	Counts DCU hardware prefetcher data read requests.
PARTIAL_STRM_ST	14	Streaming store requests.
ANY	15	Any request that crosses IDI, including I/O.

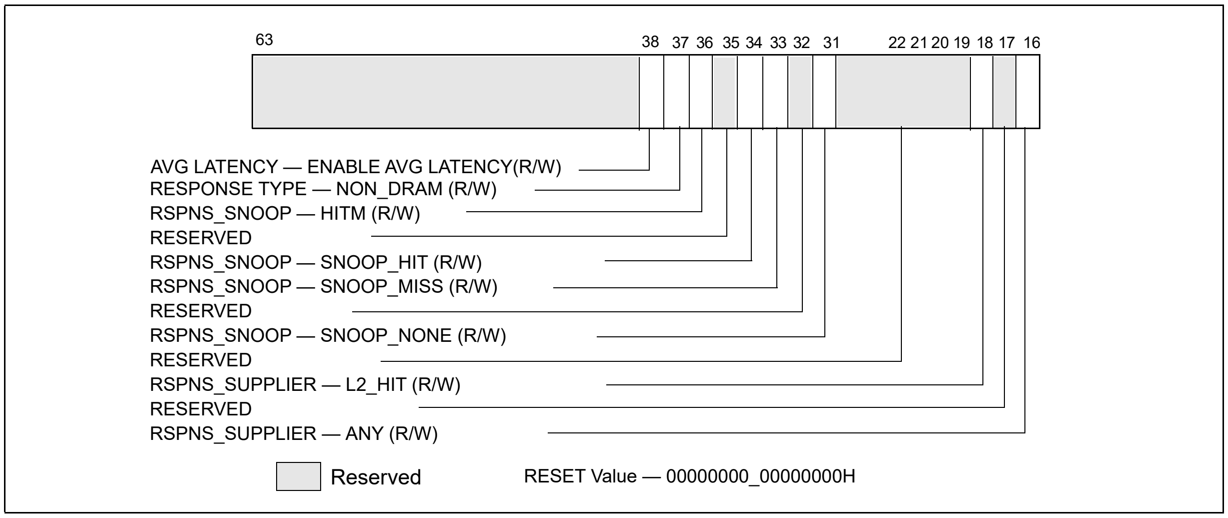

63
38 37 36 35 34 33 32 31
22 21 20 19 18 17 16
AVG LATENCY — ENABLE AVG LATENCY(R/W)
RESPONSE TYPE — NON_DRAM (R/W)
RSPNS_SNOOP — HITM (R/W)
RESERVED
RSPNS_SNOOP — SNOOP_HIT (R/W)
RSPNS_SNOOP — SNOOP_MISS (R/W)
RESERVED
RSPNS_SNOOP — SNOOP_NONE (R/W)
RESERVED
RSPNS_SUPPLIER — L2_HIT (R/W)
RESERVED
RSPNS_SUPPLIER — ANY (R/W)
Reserved
RESET Value — 00000000_00000000H

Figure 21-43. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSPx

To properly program this extra register, software must set at least one request type bit (Table 21-67) and a valid response type pattern (Table 21-68, Table 21-69). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 21-68. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	Cache reference hit L2 in either M/E/S states.
	Reserved	30:19	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]

If "ANY" bit is set, the supplier and snoop info bits are ignored.

Table 21-69. MSR_OFFCORE_RSPx Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	No details on snoop-related information.
	Reserved	32	Reserved
	SNOOP_MISS	33	Counts the number of snoop misses when L2 misses.
	SNOOP_HIT	34	Counts the number of snoops hit in the other module where no modified copies were found.
	Reserved	35	Reserved

Table 21-69. MSR_OFFCORE_RSPx Snoop Info Field Definition (Contd.)

Subtype	Bit Name	Offset	Description
	HITM	36	Counts the number of snoops hit in the other module where modified copies were found in other core's L1 cache.
	NON_DRAM	37	Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	Enable average latency measurement by counting weighted cycles of outstanding offcore requests of the request type specified in bits 15:0 and any response (bits 37:16 cleared to 0). This bit is available in MSR_OFFCORE_RESP0. The weighted cycles is accumulated in the specified programmable counter IA32_PMCx and the occurrence of specified requests are counted in the other programmable counter.

21.5.2.3 Average Offcore Request Latency Measurement

Average latency for offcore transactions can be determined by using both MSR_OFFCORE_RSP registers. Using two performance monitoring counters, program the two OFFCORE_RESPONSE event encodings into the corresponding IA32_PERFEVTSELx MSRs. Count the weighted cycles via MSR_OFFCORE_RSP0 by programming a request type in MSR_OFFCORE_RSP0.[15:0] and setting MSR_OFFCORE_RSP0.OUTSTANDING[38] to 1, white setting the remaining bits to 0. Count the number of requests via MSR_OFFCORE_RSP1 by programming the same request type from MSR_OFFCORE_RSP0 into MSR_OFFCORE_RSP1[bit 15:0], and setting MSR_OFFCORE_RSP1.ANY_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be obtained by dividing the value of the IA32_PMCx register that counted weight cycles by the register that counted requests.

21.5.3 Performance Monitoring for Goldmont Microarchitecture

Intel Atom processors based on the Goldmont microarchitecture report architectural performance monitoring versionID = 4 (see Section 21.2.4) and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 21.2.4.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 21-6 and described in Section and Section 21.2.3. The Goldmont microarchitecture does not support Hyper-Threading and thus architectural and non-architectural performance monitoring events ignore the AnyThread qualification regardless of its setting in the IA32_PERFEVTSELx MSR. However, Goldmont does not set the AnyThread deprecation bit (CPUID.0AH:EDX[15]).

The core PMU's capability is similar to that of the Silvermont microarchitecture described in Section 21.5.2, with some differences and enhancements summarized in Table 21-70.

Table 21-70. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	Goldmont Microarchitecture	Silvermont Microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 21.2.1.
# of general-purpose counters per core	4	2	Use CPUID to determine # of counters. See Section 21.2.1.
Counter width (R,W)	R:48, W: 32/48	R:40, W:32	See Section 21.2.2.
Architectural Performance Monitoring version ID	4	3	Use CPUID to determine # of counters. See Section 21.2.1.

Table 21-70. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	Goldmont Microarchitecture	Silvermont Microarchitecture	Comment
PMI Overhead Mitigation	▪ Freeze_PerfMon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics for branch profiling.	▪ Freeze_PerfMon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.	See Section 19.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 21.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz	▪ Individual counter overflow ▪ PEBS buffer overflow	See Section 21.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz, ▪ LBR_Frz	No	See Section 21.2.4.1.
PerfMon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	No	See Section 21.2.4.3.
Processor Event Based Sampling (PEBS) Events	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 21-71.	See Section 21.5.2.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 21-64).	IA32_PMC0 only.
PEBS record format encoding	0011b	0010b	
Reduce skid PEBS	IA32_PMC0 only	No	
Data Address Profiling	Yes	No	
PEBS record layout	Table 21-72; enhanced fields at offsets 90H- 98H; and TSC record field at C0H.	Table 21-65.	
PEBS EventingIP	Yes	Yes	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register.	MSR 1A6H and 1A7H, shared by a pair of cores.	Nehalem supports 1A6H only.

21.5.3.1 Processor Event Based Sampling (PEBS)

Processor event based sampling (PEBS) on the Goldmont microarchitecture is enhanced over prior generations with respect to sampling support of precise events and non-precise events. In the Goldmont microarchitecture, PEBS is supported using IA32_PMC0 for all events (see Section 19.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor at the time the sample was generated.

Precise events work the same way on Goldmont microarchitecture as on the Silvermont microarchitecture. The record will be generated after an instruction that causes the event when the counter is already overflowed and will capture the architectural state at this point (see Section 21.6.2.4 and Section 19.4.9). The eventingIP in the record will indicate the instruction that caused the event. The list of precise events supported in the Goldmont microarchitecture is shown in Table 21-71.

In the Goldmont microarchitecture, the PEBS facility also supports the use of non-precise events to record processor state information into PEBS records with the same format as with precise events.

However, a non-precise event may not be attributable to a particular retired instruction or the time of instruction execution. When the counter overflows, a PEBS record will be generated at the next opportunity. Consider the event ICACHE.HIT. When the counter overflows, the processor is fetching future instructions. The PEBS record will be generated at the next opportunity and capture the state at the processor's current retirement point. It is likely that the instruction fetch that caused the event to increment was beyond that current retirement point. Other examples of non-precise events are CPU_CLK_UNHALTED.CORE_P and HARDWARE_INTERRUPTS.RECEIVED. CPU_CLK_UNHALTED.CORE_P will increment each cycle that the processor is awake. When the counter over-flows, there may be many instructions in various stages of execution. Additionally, zero, one or multiple instructions may be retired the cycle that the counter overflows. HARDWARE_INTERRUPTS.RECEIVED increments independent of any instructions being executed. For all non-precise events, the PEBS record will be generated at the next opportunity, after the counter has overflowed. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record for a non-precise event, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked and without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32_PERF_GLOBAL_STATUS_SET.

Table 21-71. Precise Events Supported by the Goldmont Microarchitecture

Event Name	Event Select	Sub-event	UMask
LD_BLOCKS	03H	DATA_UNKNOWN	01H
		STORE_FORWARD	02H
		4K_ALIAS	04H
		UTLB_MISS	08H
		ALL_BLOCK	10H
MISALIGN_MEM_REF	13H	LOAD_PAGE_SPLIT	02H
		STORE_PAGE_SPLIT	04H
INST_RETIRED	C0H	ANY	00H
UOPS_RETITRED	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H

Table 21-71. Precise Events Supported by the Goldmont Microarchitecture (Contd.)

Event Name	Event Select	Sub-event	UMask
MEM_UOPS_RETIRED	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRED	D1H	L1_HIT	01H
		L2_HIT	02H
		L1_MISS	08H
		L2_MISS	10H
		HITM	20H
		WCB_HIT	40H
		DRAM_HIT	80H

The PEBS record format supported by processors based on the Goldmont microarchitecture is shown in Table 21-72, and each field in the PEBS record is 64 bits long.

Table 21-72. PEBS Record Format for the Goldmont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counters
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Reserved
40H	R/EBP	A8H	Reserved
48H	R/ESP	B0H	EventingRIP
50H	R8	B8H	Reserved
58H	R9	C0H	TSC
60H	R10		

On Goldmont microarchitecture, all 64 bits of architectural registers are written into the PEBS record regardless of processor mode.

With PEBS record format encoding 0011b, offset 90H reports the "Applicable Counter" field, which indicates which counters actually requested generating a PEBS record. This allows software to correlate the PEBS record entry properly with the instruction that caused the event even when multiple counters are configured to record PEBS records and multiple bits are set in the field. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

21.5.3.1.1 PEBS Data Linear Address Profiling

Goldmont supports the Data Linear Address field introduced in Haswell. It does not support the Data Source Encoding or Latency Value fields that are also part of Data Address Profiling; those fields are present in the record but are reserved.

For Goldmont microarchitecture, the Data Linear Address field will record the linear address of memory accesses in the previous instruction (e.g., the one that triggered a precise event that caused the PEBS record to be generated). Goldmont microarchitecture may record a Data Linear Address for the instruction that caused the event even for events not related to memory accesses. This may differ from other microarchitectures.

21.5.3.1.2 Reduced Skid PEBS

Processors based on Goldmont Plus microarchitecture support the Reduced Skid PEBS feature described in Section 21.9.4 on the IA32_PMC0 counter. Although Extended PEBS adds support for generating PEBS records for precise events on additional general-purpose and fixed-function performance counters, those counters do not support the Reduced Skid PEBS feature.

21.5.3.1.3 Enhancements to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62]

In addition to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62] being set when PEBS_Index reaches the PEBS_Interrupt_Theshold, the bit is also set when PEBS_Index is out of bounds. That is, the bit will be set when PEBS_Index < PEBS_Buffer_Base or PEBS_Index > PEBS_Absolute_Maximum. Note that when an out of bound condition is encountered, the overflow bits in IA32_PERF_GLOBAL_STATUS will be cleared according to Applicable Counters, however the IA32_PMCx values will not be reloaded with the Reset values stored in the DS_AREA.

21.5.3.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 21-66 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

The Goldmont microarchitecture provides unique pairs of MSR_OFFCORE_RSPx registers per core.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as follows:

- Bits 15:0 specifies the request type of a transaction request to the uncore. This is described in Table 21-73.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 21-68.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 21-74.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 21.5.2.3 for details.

Table 21-73. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	Counts demand instruction cacheline and I-side prefetch requests that miss the instruction cache.
COREWB	3	Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.

Table 21-73. MSR_OFFCORE_RSPx Request_Type Field Definition (Contd.)

Bit Name	Offset	Description
PARTIAL_READS	7	Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	Counts cacheline requests due to software prefetch instructions.
PF_L1_DATA_RD	13	Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	Counts partial cacheline writes due to streaming stores.
ANY_REQUEST	15	Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 21-67) and a valid response type pattern (either Table 21-68 or Table 21-74). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 21-74. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests

Subtype	Bit Name	Offset	Description
L2_MISS (Snoop Info)	Reserved	32:31	Reserved
	L2_MISS.SNOOP_MISS_OR_NO_SNOOP_NEEDED	33	A true miss to this module, for which a snoop request missed the other module or no snoop was performed/needed.
	L2_MISS.HIT_OTHER_CORE_NO_FWD	34	A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_CORE	36	Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	Target was a non-DRAM system address. This includes MMIO transactions.
Outstanding requests[1]	OUTSTANDING	38	Counts weighted cycles of outstanding offcore requests of the request type specified in bits 15:0, from the time the XQ receives the request and any response is received. Bits 37:16 must be set to 0. This bit is only available in MSR_OFFCORE_RESP0.

NOTES:

1. See Section 21.5.2.3, "Average Offcore Request Latency Measurement," for details on how to use this bit to extract average latency.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

Any_Response Bit | L2 Hit | 'OR' of Snoop Info Bits | Outstanding Bit

21.5.3.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 21.5.2.3.

21.5.4 Performance Monitoring for Goldmont Plus Microarchitecture

Intel Atom processors based on the Goldmont Plus microarchitecture report architectural performance monitoring versionID = 4 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 21.2.4.

Goldmont Plus performance monitoring capabilities are similar to Goldmont capabilities. The differences are in specific events and in which counters support PEBS. Goldmont Plus introduces the ability for fixed performance monitoring counters to generate PEBS records.

Goldmont Plus will set the AnyThread deprecation CPUID bit (CPUID.0AH:EDX[15]) to indicate that the Any-Thread bits in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL have no effect.

The core PMU's capability is similar to that of the Goldmont microarchitecture described in Section 21.6.3, with some differences and enhancements summarized in Table 21-75.

Table 21-75. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures

Box	Goldmont Plus Microarchitecture	Goldmont Microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 21.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 21.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change.
Architectural Performance Monitoring version ID	4	4	No change.
Processor Event Based Sampling (PEBS) Events	All General-Purpose and Fixed counters. Each General-Purpose counter supports all events (precise and non-precise).	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 21-71.	Goldmont Plus supports PEBS on all counters.
PEBS record format encoding	0011b	0011b	No change.

21.5.4.1 Extended PEBS

The PEBS facility in Goldmont Plus microarchitecture provides a number of enhancements relative to PEBS in processors from previous generations. Enhancement of PEBS facility with the Extended PEBS feature are described in detail in section 18.9.

21.5.5 Performance Monitoring for Tremont Microarchitecture

Intel Atom processors based on the Tremont microarchitecture report architectural performance monitoring versionID = 5 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 5 capabilities are described in Section 21.2.5.

Tremont performance monitoring capabilities are similar to Goldmont Plus capabilities, with the following extensions:

- Support for Adaptive PEBS.
- Support for PEBS output to Intel® Processor Trace.
- Precise Distribution support on Fixed Counter0.
- Compatibility enhancements to off-core response MSRs, MSR_OFFCORE_RSPx.

The differences and enhancements between Tremont microarchitecture and Goldmont Plus microarchitecture are summarized in Table 21-76.

Table 21-76. Core PMU Comparison Between the Tremont and Goldmont Plus Microarchitectures

Box	Tremont Microarchitecture	Goldmont Plus Microarchitecture	Comment
# of fixed counters per core	3	3	Use CPUID to determine # of counters. See Section 21.2.1.
# of general-purpose counters per core	4	4	Use CPUID to determine # of counters. See Section 21.2.1.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change. See Section 21.2.2.
Architectural Performance Monitoring version ID	5	4	
PEBS record format encoding	0100b	0011b	See Section 21.6.2.4.2.
Reduce skid PEBS	IA32_PMC0 and IA32_FIXED_CTR0	IA32_PMC0 only	
Extended PEBS	Yes	Yes	See Section 21.5.4.1.
Adaptive PEBS	Yes	No	See Section 21.9.2.
PEBS output	DS Save Area or Intel® Processor Trace	DS Save Area only	See Section 21.5.5.2.1.
PEBS record layout	See Section 21.9.2.3 for output to DS, Section 21.5.5.2.2 for output to Intel PT.	Table 21-72; enhanced fields at offsets 90H- 98H; and TSC record field at C0H.	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register, extended request and response types.	MSR 1A6H and 1A7H, each core has its own register.	

21.5.5.1 Adaptive PEBS

The PEBS record format and configuration interface has changed versus Goldmont Plus, as the Tremont microarchitecture includes support for the configurable Adaptive PEBS records; see Section 21.9.2.

21.5.5.2 PEBS output to Intel® Processor Trace

Intel Atom processors based on the Tremont microarchitecture introduce the following Precise Event-Based Sampling (PEBS) extensions:

- A mechanism to direct PEBS output into the Intel® Processor Trace (Intel® PT) output stream. In this scenario, the PEBS record is written in packetized form, in order to co-exist with other Intel PT trace data.
- New Performance Monitoring counter reload MSRs, which are used by PEBS in place of the counter reload values stored in the DS Management area when PEBS output is directed into the Intel PT output stream.

Processors that indicate support for Intel PT by setting CPUID.07H.0.EBX[25]=1, and set the new IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16] bit, support these extensions.

21.5.5.2.1 PEBS Configuration

PEBS output to Intel Processor Trace includes support for two new fields in IA32_PEBS_ENABLE.

Table 21-77. New Fields in IA32_PEBS_ENABLE

Field	Description
PMI_AFTER_EACH_RECORD[60]	Pend a PerfMon Interrupt (PMI) after each PEBS event.
PEBS_OUTPUT[62:61]	Specifies PEBS output destination. Encodings: 00B: DS Save Area. Matches legacy PEBS behavior, output location defined by IA32_DS_AREA. 01B: Intel PT trace output. 10B: Reserved. 11B: Reserved.

When PEBS_OUTPUT is set to 01B, the DS Management Area is not used and need not be configured. Instead, the output mechanism is configured through IA32_RTIT_CTL and other Intel PT MSRs, while counter reload values are configured in the MSR_RELOAD_PMCx MSRs. Details on configuring Intel PT can be found in Section 34.2.7.

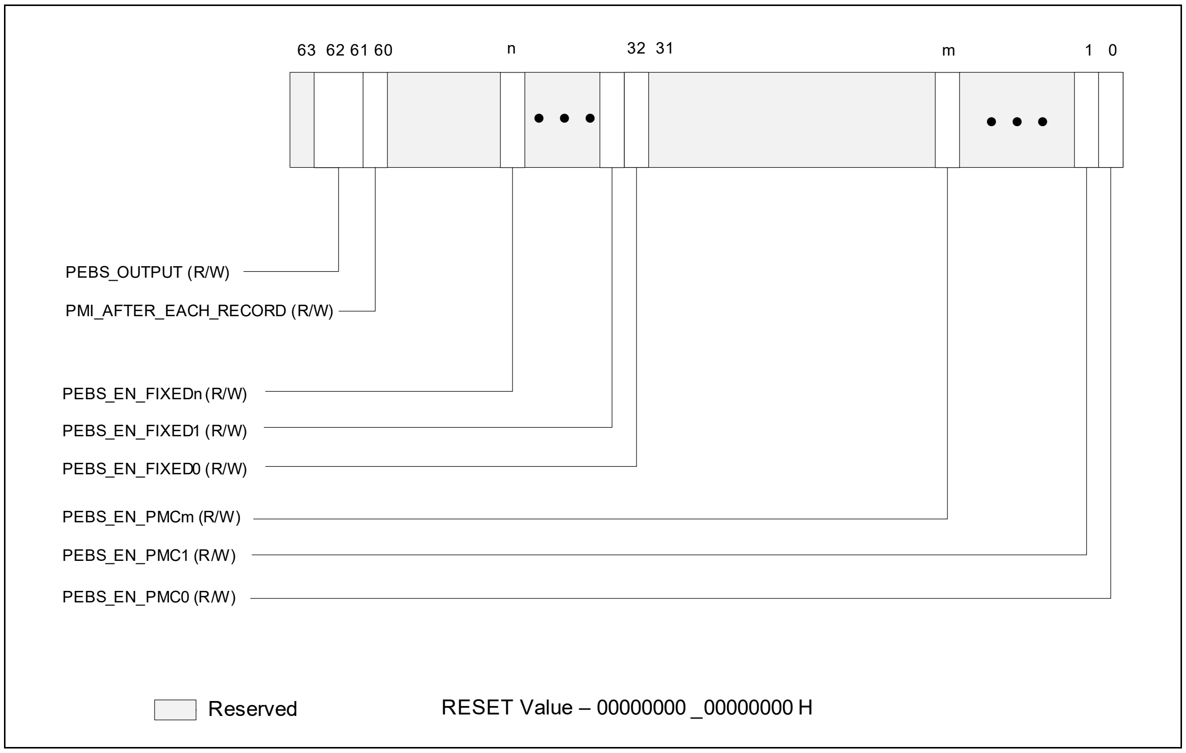


Figure 21-44. IA32_PEBS_ENABLE MSR with PEBS Output to Intel® Processor Trace

21.5.5.2.2 PEBS Record Format in Intel® Processor Trace

The format of the PEBS record changes when output to Intel PT, as the PEBS state is packetized. Each PEBS grouping is emitted as a Block Begin (BBP) and following Block Item (BIP) packets. A PEBS grouping ends when either a new PEBS grouping begins (indicated by a BBP packet) or a Block End (BEP) packet is encountered. See Section 34.4.1.1 for details of these Intel PT packets.

Because the packet headers describe the state held in the packet payload, PEBS state ordering is not fixed. PEBS state groupings may be emitted in any order, and the PEBS state elements within those groupings may be emitted in any order. Further, there is no packet that provides indication of "Record Format" or "Record Size".

If Intel PT tracing is not enabled (IA32_RTIT_STATUS.TriggerEn=0), any PEBS records triggered will be dropped. PEBS packets do not depend on ContextEn or FilterEn in IA32_RTIT_STATUS, any filtering of PEBS must be enabled from within the PerfMon configuration. Counter reload will occur in all scenarios where PEBS is triggered, regardless of TriggerEn.

The PEBS threshold mechanism for generating PerfMon Interrupts (PMIs) is not available in this mode. However, there exist other means to generate PMIs based on PEBS output. When the Intel PT ToPA output mechanism is chosen, a PMI can optionally be pended when a ToPA region is filled; see Section 34.2.7.2 for details. Further, software can opt to generate a PMI on each PEBS record by setting the new IA32_PEBS_ENABLE.PMI_AFTER_EACH_RECORD[60] bit.

The IA32_PERF_GLOBAL_STATUS.OvfDSBuffer bit will not be set in this mode.

21.5.5.2.3 PEBS Counter Reload

When PEBS output is directed into Intel PT (IA32_PEBS_ENABLE.PEBS_OUTPUT = 01B), new MSR_RELOAD_PMCx MSRs are used by the PEBS routine to reload PerfMon counters. The value from the associated reload MSR will be loaded to the appropriate counter on each PEBS event.

21.5.5.3 Precise Distribution Support on Fixed Counter 0

The Tremont microarchitecture supports the PDIR (Precise Distribution of Retired Instructions) facility, as described in Section 21.3.4.4.4, on Fixed Counter 0. Fixed Counter 0 counts the INST_RETIRED.ALL event. PEBS skid for Fixed Counter 0 will be precisely one instruction.

This is in addition to the reduced skid PEBS behavior on IA32_PMC0; see Section 21.5.3.1.2.

21.5.5.4 Compatibility Enhancements to Offcore Response MSRs

The Off-core Response facility is similar to that described in Section 21.5.3.2.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as shown below. RequestType bits are defined in Table 21-78, ResponseType bits in Table 21-79, and SnoopInfo bits in Table 21-80.

Table 21-78. MSR_OFFCORE_RSPx Request Type Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	Counts demand data reads.
DEMAND_RFO	1	Counts all demand reads for ownership (RFO) requests and software based prefetches for exclusive ownership (prefetchw).
DEMAND_CODE_RD	2	Counts demand instruction fetches and L1 instruction cache prefetches.
COREWB_M	3	Counts modified write backs from L1 and L2.
HWPF_L2_DATA_RD	4	Counts prefetch (that bring data to L2) data reads.
HWPF_L2_RFO	5	Counts all prefetch (that bring data to L2) RFOs.
HWPF_L2_CODE_RD	6	Counts all prefetch (that bring data to L2 only) code reads.
Reserved	9:7	Reserved.
HWPF_L1D_AND_SWPF	10	Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).
STREAMING_WR	11	Counts all streaming stores.
COREWB_NONM	12	Counts non-modified write backs from L2.
Reserved	14:13	Reserved.
OTHER	15	Counts miscellaneous requests, such as I/O accesses that have any response type.
UC_RD	44	Counts uncached memory reads (PRd, UCRdF).
UC_WR	45	Counts uncached memory writes (WiL).
PARTIAL_STREAMING_WR	46	Counts partial (less than 64 byte) streaming stores (WCiL).
FULL_STREAMING_WR	47	Counts full, 64 byte streaming stores (WCiLF).

Table 21-78. MSR_OFFCORE_RSPx Request Type Definition (Contd.)

Bit Name	Offset	Description
L1WB_M	48	Counts modified WriteBacks from L1 that miss the L2.
L2WB_M	49	Counts modified WriteBacks from L2.

Table 21-79. MSR_OFFCORE_RSPx Response Type Definition

Bit Name	Offset	Description
ANY_RESPONSE	16	Catch all value for any response types.
L3_HIT_M	18	LLC/L3 Hit - M-state.
L3_HIT_E	19	LLC/L3 Hit - E-state.
L3_HIT_S	20	LLC/L3 Hit - S-state.
L3_HIT_F	21	LLC/L3 Hit - I-state.
LOCAL_DRAM	26	LLC/L3 Miss, DRAM Hit.
OUTSTANDING	63	Average latency of outstanding requests with the other counter counting number of occurrences; can also can be used to count occupancy.

Table 21-80. MSR_OFFCORE_RSPx Snoop Info Definition

Bit Name	Offset	Description
SNOOP_NONE	31	None of the cores were snooped. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_NOT_NEEDED	32	No snoop needed to satisfy the request. ▪ LLC hit and CV bit(s) (core valid) was not set. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_MISS	33	A snoop was sent but missed. ▪ LLC hit and CV bit(s) was set but snoop missed (silent data drop in core), data returned from LLC. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_HIT_NO_FWD	34	A snoop was sent but no data forward. ▪ LLC hit and CV bit(s) was set but no data forward from the core, data returned from LLC. ▪ LLC miss and Dram data returned directly to the core.
SNOOP_HIT_WITH_FWD	35	A snoop was sent and non-modified data was forward. ▪ LLC hit and CV bit(s) was set, non-modified data was forward from core.
SNOOP_HITM	36	A snoop was sent and modified data was forward. ▪ LLC hit E or M and the CV bit(s) was set, modified data was forward from core.
NON_DRAM_BIT	37	Target was non-DRAM system address, MMIO access. ▪ LLC miss and Non-Dram data returned.

The Off-core Response capability behaves as follows:

- To specify a complete offcore response filter, software must properly program at least one RequestType and one ResponseType. A valid request type must have at least one bit set in the non-reserved bits of 15:0 or 49:44. A valid response type must be a non-zero value of one the following expressions:
    - Read requests:

      Any_Response Bit | (‘OR’ of Supplier Info Bits) ‘AND’ (‘OR’ of Snoop Info Bits) | Outstanding Bit
    - Write requests:

      Any_Response Bit | (‘OR’ of Supplier Info Bits) | Outstanding Bit
- When the ANY_RESPONSE bit in the ResponseType is set, all other response type bits will be ignored.
- True Demand Cacheable Loads include neither L1 Prefetches nor Software Prefetches.
- Bits 15:0 and Bits 49:44 specifies the request type of a transaction request to the uncore. This is described in Table 21-78.
- Bits 30:16 specifies common supplier information.
- “Outstanding Requests” (bit 63) is only available on MSR_OFFCORE_RSP0; a #GP fault will occur if software attempts to write a 1 to this bit in MSR_OFFCORE_RSP1. It is mutually exclusive with any ResponseType. Software must guarantee that all other ResponseType bits are set to 0 when the “Outstanding Requests” bit is set.
- “Outstanding Requests” bit 63 can enable measurement of the average latency of a specific type of off-core transaction; two programmable counters must be used simultaneously and the RequestType programming for MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 must be the same when using this Average Latency feature. See Section 21.5.2.3 for further details.

21.6 PERFORMANCE MONITORING (LEGACY INTEL PROCESSORS)

21.6.1 Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)

In Intel Core Solo and Intel Core Duo processors, non-architectural performance monitoring events are programmed using the same facilities (see Figure 21-1) used for architectural performance events.

Non-architectural performance events use event select values that are model-specific. Event mask (Umask) values are also specific to event logic units. Some microarchitectural conditions detectable by a Umask value may have specificity related to processor topology (see Section 10.6, “Detecting Hardware Multi-Threading Support and Topology,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). As a result, the unit mask field (for example, IA32_PERFEVTSELx[bits 15:8]) may contain sub-fields that specify topology information of processor cores.

The sub-field layout within the Umask field may support two-bit encoding that qualifies the relationship between a microarchitectural condition and the originating core. This data is shown in Table 21-81. The two-bit encoding for core-specificity is only supported for a subset of Umask values (see: https://perfmon-events.intel.com/) and for Intel Core Duo processors. Such events are referred to as core-specific events.

Table 21-81. Core Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 15:14 Encoding	**Description**
11B	All cores
10B	Reserved
01B	This core
00B	Reserved

Some microarchitectural conditions allow detection specificity only at the boundary of physical processors. Some bus events belong to this category, providing specificity between the originating physical processor (a bus agent) versus other agents on the bus. Sub-field encoding for agent specificity is shown in Table 21-82.

Table 21-82. Agent Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13 Encoding	**Description**
0	This agent
1	Include all agents

Some microarchitectural conditions are detectable only from the originating core. In such cases, unit mask does not support core-specificity or agent-specificity encodings. These are referred to as core-only conditions.

Some microarchitectural conditions allow detection specificity that includes or excludes the action of hardware prefetches. A two-bit encoding may be supported to qualify hardware prefetch actions. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 21-83.

Table 21-83. HW Prefetch Qualification Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13:12 Encoding	**Description**
11B	All inclusive
10B	Reserved
01B	Hardware prefetch only
00B	Exclude hardware prefetch

Some performance events may (a) support none of the three event-specific qualification encodings (b) may support core-specificity and agent specificity simultaneously (c) or may support core-specificity and hardware prefetch qualification simultaneously. Agent-specificity and hardware prefetch qualification are mutually exclusive.

In addition, some L2 events permit qualifications that distinguish cache coherent states. The sub-field definition for cache coherency state qualification is shown in Table 21-84. If no bits in the MESI qualification sub-field are set for an event that requires setting MESI qualification bits, the event count will not increment.

Table 21-84. MESI Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	Counts modified state
Bit 10	Counts exclusive state
Bit 9	Counts shared state
Bit 8	Counts Invalid state

21.6.2 Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)

In addition to architectural performance monitoring, processors based on the Intel Core microarchitecture support non-architectural performance monitoring events.

Architectural performance events can be collected using general-purpose performance counters. Non-architectural performance events can be collected using general-purpose performance counters (coupled with two IA32_PERFE-VTSELx MSRs for detailed event configurations), or fixed-function performance counters (see Section 21.6.2.1). IA32_PERFEVTSELx MSRs are architectural; their layout is shown in Figure 21-1. Starting with Intel Core 2 processor T 7700, fixed-function performance counters and associated counter control and status MSR becomes part of architectural performance monitoring version 2 facilities (see also Section 21.2.2).

Non-architectural performance events in processors based on Intel Core microarchitecture use event select values that are model-specific. Valid event mask (Umask) bits can be found at: https://perfmon-events.intel.com/. The UMASK field may contain sub-fields identical to those listed in Table 21-81, Table 21-82, Table 21-83, and Table 21-84. One or more of these sub-fields may apply to specific events on an event-by-event basis.

In addition, the UMASK filed may also contain a sub-field that allows detection specificity related to snoop responses. Bits of the snoop response qualification sub-field are defined in Table 21-85.

Table 21-85. Bus Snoop Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	HITM response
Bit 10	Reserved
Bit 9	HIT response
Bit 8	CLEAN response

There are also non-architectural events that support qualification of different types of snoop operation. The corresponding bit field for snoop type qualification are listed in Table 21-86.

Table 21-86. Snoop Type Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 9:8	**Description**
Bit 9	CMP2I snoops
Bit 8	CMP2S snoops

No more than one sub-field of MESI, snoop response, and snoop type qualification sub-fields can be supported in a performance event.

NOTE

Software must write known values to the performance counters prior to enabling the counters. The content of general-purpose counters and fixed-function counters are undefined after INIT or RESET.

21.6.2.1 Fixed-function Performance Counters

Processors based on Intel Core microarchitecture provide three fixed-function performance counters. Bits beyond the width of the fixed counter are reserved and must be written as zeros. Model-specific fixed-function performance counters on processors that support Architectural PerfMon version 1 are 40 bits wide.

Each of the fixed-function counter is dedicated to count a pre-defined performance monitoring events. See Table 21-1 for details of the PMC addresses and what these events count.

Programming the fixed-function performance counters does not involve any of the IA32_PERFEVTSELx MSRs, and does not require specifying any event masks. Instead, the MSR IA32_FIXED_CTR_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 21-45. Two sub-fields are defined for each control. See Figure 21-45; bit fields are:

- **Enable field (low 2 bits in each 4-bit control) —** When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring 0.

When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring greater than 0.

Writing 0 to both bits stops the performance counter. Writing 11B causes the counter to increment irrespective of privilege levels.

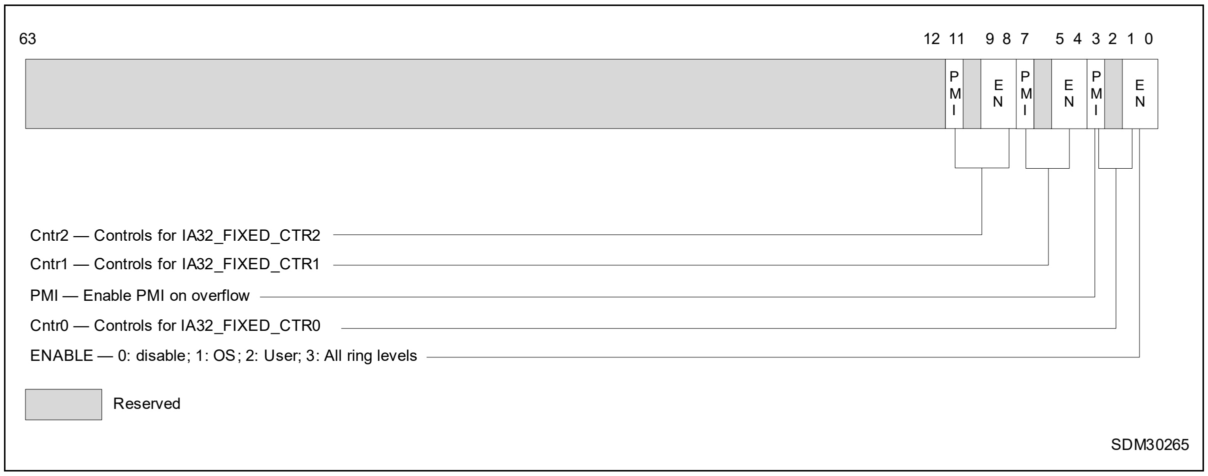

63
12 11 9 8 7 5 4 3 2 1 0
PMI
EN
PMI
EN
PMI
EN
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow
Cntr0 — Controls for IA32_FIXED_CTR0
ENABLE — 0: disable; 1: OS; 2: User; 3: All ring levels
Reserved
SDM30265

Figure 21-45. Layout of IA32_FIXED_CTR_CTRL MSR

- **PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

21.6.2.2 Global Counter Control Facilities

Processors based on Intel Core microarchitecture provides simplified performance counter control that simplifies the most frequent operations in programming performance events, i.e., enabling/disabling event counting and checking the status of counter overflows. This is done by the following three MSRs:

- MSR_PERF_GLOBAL_CTRL enables/disables event counting for all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
- MSR_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single RDMSR.
- MSR_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.

MSR_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 21-46). Each enable bit in MSR_PERF_GLOBAL_CTRL is ANDed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the ANDed results is true; counting is disabled when the result is false.

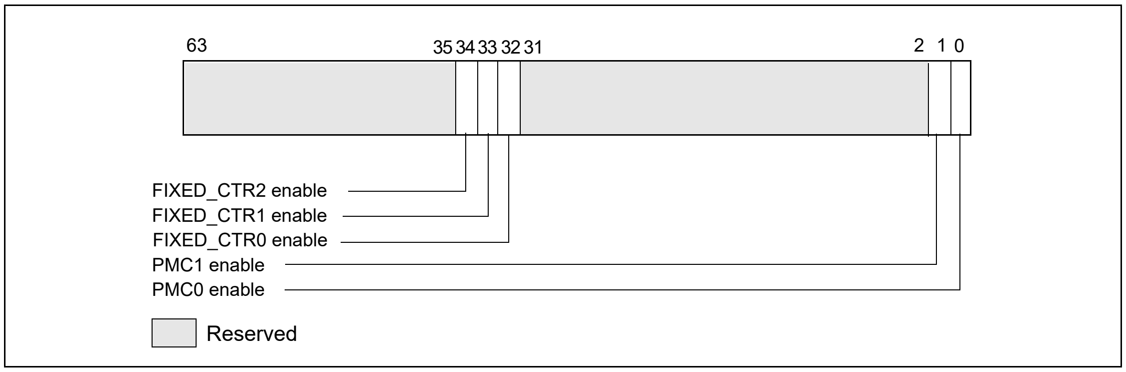


Figure 21-46. Layout of MSR_PERF_GLOBAL_CTRL MSR

MSR_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. MSR_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. MSR_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware (see Figure 21-47). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has occurred in the associated counter.

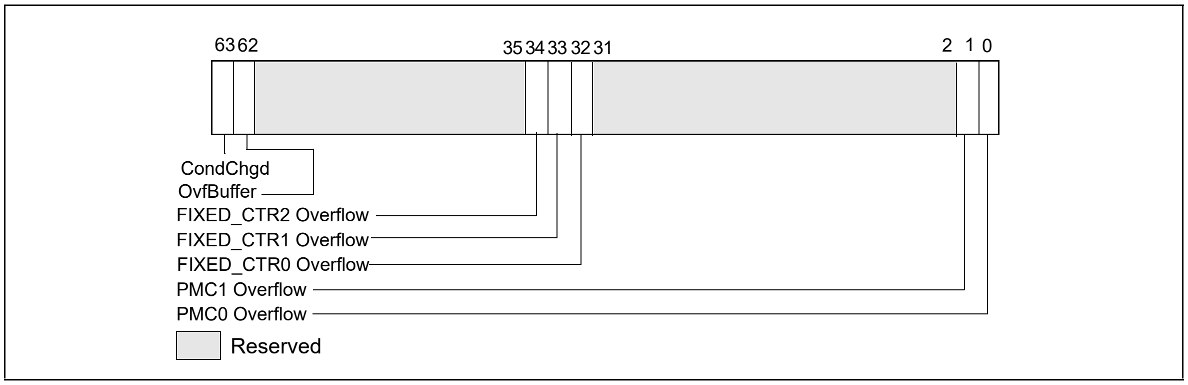


Figure 21-47. Layout of MSR_PERF_GLOBAL_STATUS MSR

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 19.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

MSR_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 21-48). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

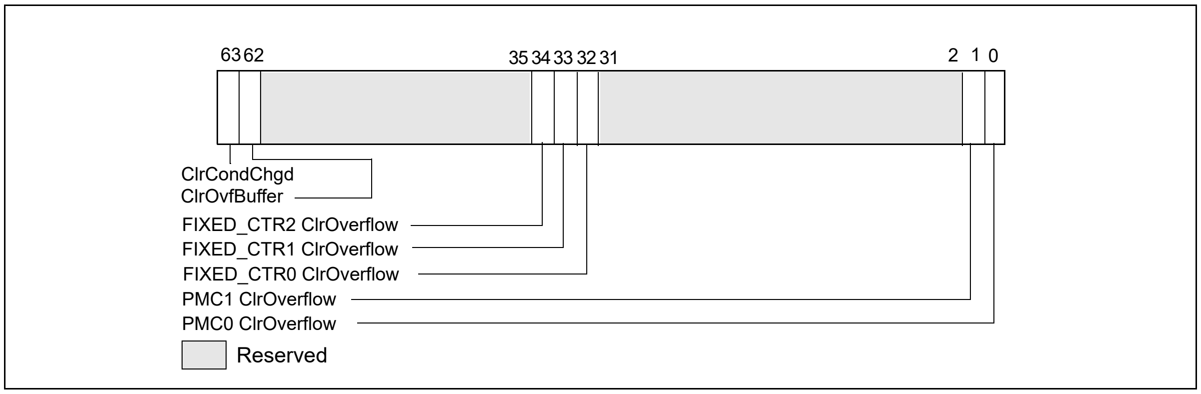


Figure 21-48. Layout of MSR_PERF_GLOBAL_OVF_CTRL MSR

21.6.2.3 At-Retirement Events

Many non-architectural performance events are impacted by the speculative nature of out-of-order execution. A subset of non-architectural performance events on processors based on Intel Core microarchitecture are enhanced with a tagging mechanism (similar to that found in Intel NetBurst® microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core microarchitecture does not require special MSR programming control (see Section 21.6.3.6, "At-Retirement Counting"), but is limited to IA32_PMC0. See Table 21-87 for a list of events available to processors based on Intel Core microarchitecture.

Table 21-87. At-Retirement Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
ITLB_MISS_RETIRED	00H	C9H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

21.6.2.4 Processor Event Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support processor event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 21.6.2.4.2 and Section 19.4.9).

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, precise events that can be used with PEBS are listed in Table 21-88. The procedure for detecting availability of PEBS is the same as described in Section 21.6.3.8.1.

Table 21-88. PEBS Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

21.6.2.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32_PMC0 only. Use the following procedure to set up the processor and IA32_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 19-8 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32_PEBS_ENABLE MSR.
3. Set up the IA32_PMC0 performance counter and IA32_PERFEVTSEL0 for an event listed in Table 21-88.

21.6.2.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32_PERF_CAPABILITES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version ID equals 2 or higher. The bit fields of IA32_PERF_CAPABILITES are defined in Table 2-2 of Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. The relevant bit fields that governs PEBS are:

- PEBSTrap [bit 6]: When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.
- PEBSSaveArchRegs [bit 7]: When set, PEBS will save architectural register and state information according to the encoded value of the PEBSRecordFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1.
- PEBSRecordFormat [bits 11:8]: Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (See Section 21.6.3.8).
  - 0001B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS and load latency data. (See Section 21.3.1.1.1).
  - 0010B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS, load latency data, and TSX tuning information. (See Section 21.3.6.2).
  - 0011B: PEBS record includes additional information of load latency data, TSX tuning information, TSC data, and the applicable counter field replaces IA32_PERF_GLOBAL_STATUS at offset 90H. (See Section 21.3.8.1.1).
  - 0100B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG. (See Section 21.9.2.3). The PEBS Configuration Buffer is defined as shown in Figure 21-66 with Counter Reset fields allocation for 8 general-purpose counters followed by 4 fixed-function counters.

— 0101B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG. (See Section 21.9.2.3). The PEBS Configuration Buffer is defined as shown in Figure 21-66 with Counter Reset fields allocation for 32 general-purpose counters followed by 16 fixed-function counters.

— 0110B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG (see Figure 21-73 in Section 21.9.2.3) that is compatible with the previous MSR_PEBS_DATA_CFG (see Figure 21-72 in Section 21.9.2.3). The PEBS Config Buffer is defined as shown in Figure 21-72 with a Counter Reset fields allocation for 32 general-purpose counters followed by 16 fixed-function counters.

21.6.2.4.3 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the Interrupt-based event sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 19.4.9.1, “64 Bit Format of the DS Save Area,” for guidelines when writing the DS ISR.

The service routine can query MSR_PERF_GLOBAL_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR_PERF_GLOBAL_OVF_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 21-89.

Table 21-89. Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	**For Processors based on Intel NetBurst microarchitecture**
Verify PEBS support of processor/OS.	▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled.	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0. On subsequent entries: ▪ Clear all counters if “Counter Freeze on PMI“ is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[1]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write “sample-after“ values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable.	▪ Set appropriate OVF_PMI bits - 1. ▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

NOTES:

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

21.6.2.4.4 Re-configuring PEBS Facilities

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32_DS_AREA).

21.6.3 Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMC instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMC instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32_MISC_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCRs sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32_DS_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR_PEBS_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 21-90 lists the performance counters and their associated CCCRs, along with the ESCRs that select events to be counted for each performance counter. Predefined event metrics and events can be found at: https://perfmon-events.intel.com/.

Table 21-90. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_BPU_COUNTER0	0	300H	MSR_BPU_CCCR0	360H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H

Table 21-90. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCR0	364H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCR0	368H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER1	9	309H	MSR_FLAME_CCCR1	369H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H

Table 21-90. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0	4	3B8H
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0[1]	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0	4	3B8H
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0[1]	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1[1]	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH
MSR_IQ_COUNTER3	15	30FH	MSR_IQ_CCCR3	36FH	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1[1]	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H	MSR_CRU_ESCR0	4	3B8H
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0[1]	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1[1]	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH

NOTES:

1. MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- Non-retirement events are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

  The at-retirement counting mechanism includes facilities for tagging μops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting —** A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.
- **Interrupt-based event sampling —** A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted.

  When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.
- **Processor event-based sampling (PEBS) —** In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

21.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 21-90) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 21-90) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 21-49 shows the layout of an ESCR MSR. The functions of the flags and fields are:

- **USR flag, bit 2 —** When set, events are counted when the processor is operating at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.
- **OS flag, bit 3 —** When set, events are counted when the processor is operating at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the OS and USR flags are set, events are counted at all privilege levels.)

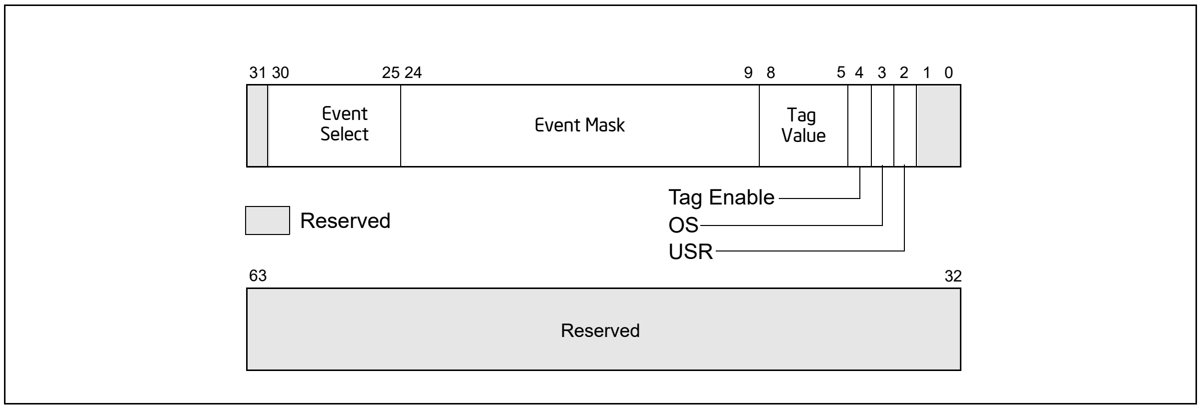


Figure 21-49. Event Selection Control Register (ESCR) for Pentium 4 and Intel® Xeon® Processors without Intel HT Technology Support

- **Tag enable, bit 4 —** When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 21.6.3.6, “At-Retirement Counting.”

- **Tag value field, bits 5 through 8 —** Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 21-90 gives the addresses of the ESCR MSRs.

Writing to an ESCR MSR does not enable counting with its associated performance counter; it only selects the event or events to be counted. The CCCR for the selected performance counter must also be configured. Configuration of the CCCR includes selecting the ESCR and enabling the counter.

21.6.3.2 Performance Counters

The performance counters in conjunction with the counter configuration control registers (CCCRs) are used for filtering and counting the events selected by the ESCRs. Processors based on Intel NetBurst microarchitecture provide 18 performance counters organized into 9 pairs. A pair of performance counters is associated with a particular subset of events and ESCR's (see Table 21-90). The counter pairs are partitioned into four groups:

- The BPU group, includes two performance counter pairs:
  - MSR_BPU_COUNTER0 and MSR_BPU_COUNTER1.
  - MSR_BPU_COUNTER2 and MSR_BPU_COUNTER3.
- The MS group, includes two performance counter pairs:
  - MSR_MS_COUNTER0 and MSR_MS_COUNTER1.
  - MSR_MS_COUNTER2 and MSR_MS_COUNTER3.
- The FLAME group, includes two performance counter pairs:
  - MSR_FLAME_COUNTER0 and MSR_FLAME_COUNTER1.
  - MSR_FLAME_COUNTER2 and MSR_FLAME_COUNTER3.
- The IQ group, includes three performance counter pairs:
  - MSR_IQ_COUNTER0 and MSR_IQ_COUNTER1.
  - MSR_IQ_COUNTER2 and MSR_IQ_COUNTER3.
  - MSR_IQ_COUNTER4 and MSR_IQ_COUNTER5.

The MSR_IQ_COUNTER4 counter in the IQ group provides support for the PEBS.

Alternate counters in each group can be cascaded: the first counter in one pair can start the first counter in the second pair and vice versa. A similar cascading is possible for the second counters in each pair. For example, within the BPU group of counters, MSR_BPU_COUNTER0 can start MSR_BPU_COUNTER2 and vice versa, and MSR_BPU_COUNTER1 can start MSR_BPU_COUNTER3 and vice versa (see Section 21.6.3.5.6, "Cascading Counters"). The cascade flag in the CCCR register for the performance counter enables the cascading of counters.

Each performance counter is 40-bits wide (see Figure 21-50). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or, if ECX[31] is set to 1, the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits. In such cases, counter bits 31:0 are written to EAX, while 0 is written to EDX.

The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

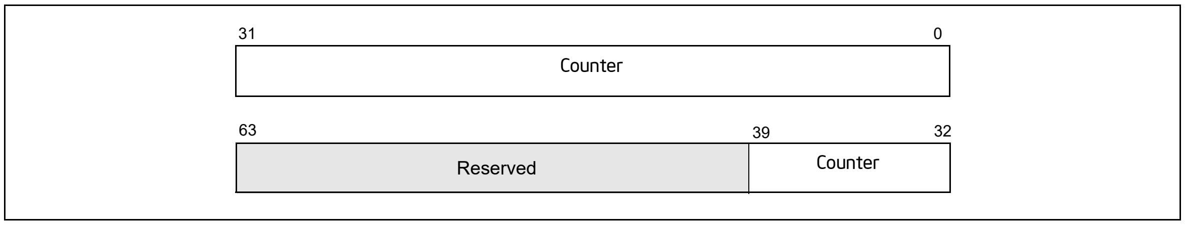


Figure 21-50. Performance Counter (Pentium 4 and Intel® Xeon® Processors)

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement negative integer in the counter. The counter will then count from the preset value up to -1 and overflow. Writing to a performance counter in a Pentium 4 or Intel Xeon processor with the WRMSR instruction causes all 40 bits of the counter to be written.

21.6.3.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 21-90). The CCCRs control the filtering and counting of events as well as interrupt generation. Figure 21-51 shows the layout of an CCCR MSR. The functions of the flags and fields are as follows:

- **Enable flag, bit 12 —** When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset.
- **ESCR select field, bits 13 through 15 —** Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Compare flag, bit 18 —** When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.
- **Complement flag, bit 19 —** Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 21.6.3.5.2, "Filtering Events"). The complement flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23 —** Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 21.6.3.5.2, "Filtering Events").
- **Edge flag, bit 24 —** When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.

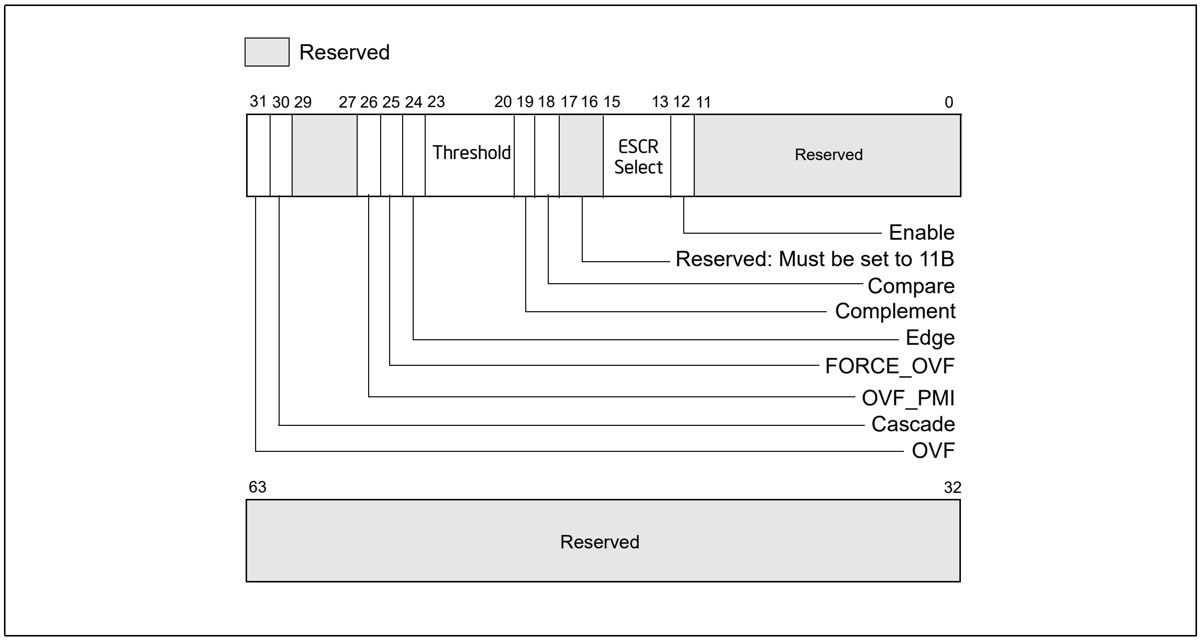


Figure 21-51. Counter Configuration Control Register (CCCR)

- **FORCE_OVF flag, bit 25 —** When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI flag, bit 26 —** When set, causes a performance monitor interrupt (PMI) to be generated when the counter overflows occurs; when clear, disables PMI generation. Note that the PMI is generated on the next event count after the counter has overflowed.
- **Cascade flag, bit 30 —** When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 21.6.3.2, "Performance Counters," for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31 —** Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

The CCCRs are initialized to all 0s on reset.

The events that an enabled performance counter actually counts are selected and filtered by the following flags and fields in the ESCR and CCCR registers and in the qualification order given:

1. The event select and event mask fields in the ESCR select a class of events to be counted and one or more event types within the class, respectively.
2. The OS and USR flags in the ESCR selected the privilege levels at which events will be counted.
3. The ESCR select field of the CCCR selects the ESCR. Since each counter has several ESCRs associated with it, one ESCR must be chosen to select the classes of events that may be counted.
4. The compare and complement flags and the threshold field of the CCCR select an optional threshold to be used in qualifying an event count.
5. The edge flag in the CCCR allows events to be counted only on rising-edge transitions.

The qualification order in the above list implies that the filtered output of one "stage" forms the input for the next. For instance, events filtered using the privilege level flags can be further qualified by the compare and complement flags and the threshold field, and an event that matched the threshold criteria, can be further qualified by edge detection.

The uses of the flags and fields in the CCCRs are discussed in greater detail in Section 21.6.3.5, "Programming the Performance Counters for Non-Retirement Events."

21.6.3.4 Debug Store (DS) Mechanism

The debug store (DS) mechanism was introduced with processors based on Intel NetBurst microarchitecture to allow various types of information to be collected in memory-resident buffers for use in debugging and tuning programs. The DS mechanism can be used to collect two types of information: branch records and processor event-based sampling (PEBS) records. The availability of the DS mechanism in a processor is indicated with the DS feature flag (bit 21) returned by the CPUID instruction.

See Section 19.4.5, "Branch Trace Store (BTS)," and Section 21.6.3.8, "Processor Event-Based Sampling (PEBS)," for a description of these facilities. Records collected with the DS mechanism are saved in the DS save area. See Section 19.4.9, "BTS and DS Save Area."

21.6.3.5 Programming the Performance Counters for Non-Retirement Events

The basic steps to program a performance counter and to count events include the following:

1. Select the event or events to be counted.
2. For each event, select an ESCR that supports the event.
3. Match the CCCR Select value and ESCR name to a value listed in Table 21-90; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which they are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

21.6.3.5.1 Selecting Events to Count

There is a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event, setup information is provided. Table 21-91 gives an example of one of the events.

Table 21-91. Event Example

Event Name	Event Parameters	Parameter Value	Description
branch_retired			Counts the retirement of a branch. Specify one or more mask bits to select any combination of branch taken, not-taken, predicted, and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 15-3 for the addresses of the ESCR MSRs.
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 15-3.
	ESCR Event Select	06H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	ESCR[24:9] Branch Not-taken Predicted Branch Not-taken Mispredicted Branch Taken Predicted Branch Taken Mispredicted
	CCCR Select	05H	CCCR[15:13]

Table 21-91. Event Example (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

Event Parameters are described below.

- **ESCR restrictions —** Lists the ESCRs that can be used to program the event. Typically only one ESCR is needed to count an event.
- **Counter numbers per ESCR —** Lists which performance counters are associated with each ESCR. Table 21-90 gives the name of the counter and CCCR for each counter number. Typically only one counter is needed to count the event.
- **ESCR event select —** Gives the value to be placed in the event select field of the ESCR to select the event.
- **ESCR event mask —** Gives the value to be placed in the Event Mask field of the ESCR to select sub-events to be counted. The parameter value column defines the documented bits with relative bit position offset starting from 0, where the absolute bit position of relative offset 0 is bit 9 of the ESCR. All undocumented bits are reserved and should be set to 0.
- **CCCR select —** Gives the value to be placed in the ESCR select field of the CCCR associated with the counter to select the ESCR to be used to define the event. This value is not the address of the ESCR; it is the number of the ESCR from the Number column in Table 21-90.
- **Event specific notes —** Gives additional information about the event, such as the name of the same or a similar event defined for the P6 family processors.
- **Can support PEBS —** Indicates if PEBS is supported for the event (only supplied for at-retirement events).
- **Requires additional MSR for tagging —** Indicates which if any additional MSRs must be programmed to count the events (only supplied for the at-retirement events).

NOTE

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The following procedure shows how to set up a performance counter for basic counting; that is, the counter is set up to count a specified event indefinitely, wrapping around whenever it reaches its maximum count. This procedure is continued through the following four sections.

An event to be counted can be selected as follows:

1. Select the event to be counted.
2. Select the ESCR to be used to select events to be counted from the ESCRs field.
3. Select the number of the counter to be used to count the event from the Counter Numbers Per ESCR field.
4. Determine the name of the counter and the CCCR associated with the counter, and determine the MSR addresses of the counter, CCCR, and ESCR from Table 21-90.
5. Use the WRMSR instruction to write the ESCR Event Select and ESCR Event Mask values into the appropriate fields in the ESCR. At the same time set or clear the USR and OS flags in the ESCR as desired.
6. Use the WRMSR instruction to write the CCCR Select value into the appropriate field in the CCCR.

NOTE

Typically all the fields and flags of the CCCR will be written with one WRMSR instruction; however, in this procedure, several WRMSR writes are used to more clearly demonstrate the uses of the various CCCR fields and flags.

This setup procedure is continued in the next section, Section 21.6.3.5.2, “Filtering Events.”

21.6.3.5.2 Filtering Events

Each counter receives up to 4 input lines from the processor hardware from which it is counting events. The counter treats these inputs as binary inputs (input 0 has a value of 1, input 1 has a value of 2, input 3 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counters threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a “greater than” or a “less than or equal to” comparison of the input value vs. a threshold value can be made. The complement flag selects “less than or equal to” (flag set) or “greater than” (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, than any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a “rising edge” event; that is, a false-to-true transition. Figure 21-52 illustrates rising edge filtering.

The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 21.6.3.5.1, “Selecting Events to Count.”

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
   - Set the compare flag.
   - Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
   - Enter a value from 0 to 15 in the threshold field.
8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 21.6.3.5.3, “Starting Event Counting.”

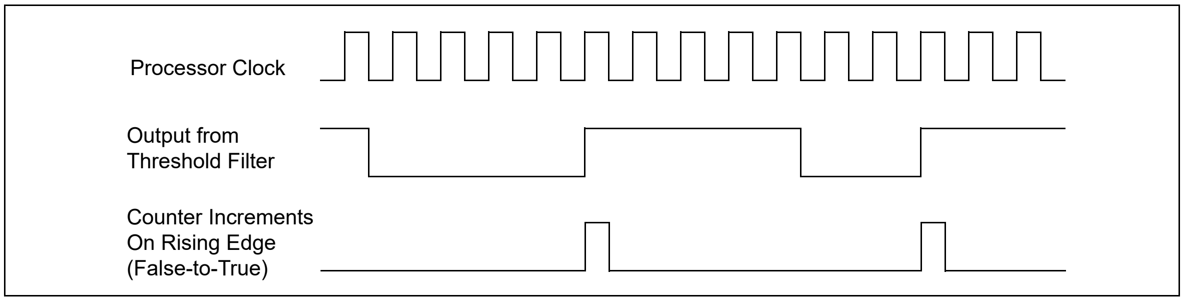

Processor Clock
Output from Threshold Filter
Counter Increments On Rising Edge (False-to-True)

Figure 21-52. Effects of Edge Filtering

21.6.3.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter's CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 21.6.3.5.5, "Halting Event Counting").

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 21.6.3.5.2, "Filtering Events."

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 21.6.3.5.4, "Reading a Performance Counter's Count."

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 21.6.3.5.6, "Cascading Counters").

21.6.3.5.4 Reading a Performance Counter's Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 21.6.3.2, "Performance Counters." These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 21.6.3.5.3, "Starting Event Counting."

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 21-90 used as an operand.

This setup procedure is continued in the next section, Section 21.6.3.5.5, "Halting Event Counting."

21.6.3.5.5 Halting Event Counting

After a performance counter has been started (enabled), it continues counting indefinitely. If the counter overflows (goes one count past its maximum count), it wraps around and continues counting. When the counter wraps around, it sets its OVF flag to indicate that the counter has overflowed. The OVF flag is a sticky flag that indicates that the counter has overflowed at least once since the OVF bit was last cleared.

To halt counting, the CCCR enable flag for the counter must be cleared.

The following procedural step shows how to stop event counting. This step is a continuation of the setup procedure introduced in Section 21.6.3.5.4, "Reading a Performance Counter's Count."

11. To stop event counting, execute a WRMSR instruction to clear the CCCR enable flag for the performance counter.

To halt a cascaded counter (a counter that was started when its alternate counter overflowed), either clear the Cascade flag in the cascaded counter's CCCR MSR or clear the OVF flag in the alternate counter's CCCR MSR.

21.6.3.5.6 Cascading Counters

As described in Section 21.6.3.2, "Performance Counters," eighteen performance counters are implemented in pairs. Nine pairs of counters and associated CCCRs are further organized as four blocks: BPU, MS, FLAME, and IQ (see Table 21-90). The first three blocks contain two pairs each. The IQ block contains three pairs of counters (12 through 17) with associated CCCRs (MSR_IQ_CCCR0 through MSR_IQ_CCCR5).

The first 8 counter pairs (0 through 15) can be programmed using ESCRs to detect performance monitoring events. Pairs of ESCRs in each of the four blocks allow many different types of events to be counted. The cascade flag in the CCCR MSR allows nested monitoring of events to be performed by cascading one counter to a second counter located in another pair in the same block (see Figure 21-51 for the location of the flag).

Counters 0 and 1 form the first pair in the BPU block. Either counter 0 or 1 can be programmed to detect an event via MSR_MO B_ESCR0. Counters 0 and 2 can be cascaded in any order, as can counters 1 and 3. It's possible to set up 4 counters in the same block to cascade on two pairs of independent events. The pairing described also applies to subsequent blocks. Since the IQ PUB has two extra counters, cascading operates somewhat differently if 16 and 17 are involved. In the IQ block, counter 16 can only be cascaded from counter 14 (not from 12); counter 14 cannot be cascaded from counter 16 using the CCCR cascade bit mechanism. Similar restrictions apply to counter 17.

Example 21-1. Counting Events

Assume a scenario where counter X is set up to count 200 occurrences of event A; then counter Y is set up to count 400 occurrences of event B. Each counter is set up to count a specific event and overflow to the next counter. In the above example, counter X is preset for a count of -200 and counter Y for a count of -400; this setup causes the counters to overflow on the 200th and 400th counts respectively.

Continuing this scenario, counter X is set up to count indefinitely and wraparound on overflow. This is described in the basic performance counter setup procedure that begins in Section 21.6.3.5.1, "Selecting Events to Count." Counter Y is set up with the cascade flag in its associated CCCR MSR set to 1 and its enable flag set to 0.

To begin the nested counting, the enable bit for the counter X is set. Once enabled, counter X counts until it overflows. At this point, counter Y is automatically enabled and begins counting. Thus counter X overflows after 200 occurrences of event A. Counter Y then starts, counting 400 occurrences of event B before overflowing. When performance counters are cascaded, the counter Y would typically be set up to generate an interrupt on overflow. This is described in Section 21.6.3.5.8, "Generating an Interrupt on Overflow."

The cascading counters mechanism can be used to count a single event. The counting begins on one counter then continues on the second counter after the first counter overflows. This technique doubles the number of event counts that can be recorded, since the contents of the two counters can be added together.

21.6.3.5.7 EXTENDED CASCADING

Extended cascading is a model-specific feature in the Intel NetBurst microarchitecture with CPUID DisplayFamily_DisplayModel 0F_02, 0F_03, 0F_04, 0F_06. This feature uses bit 11 in CCCRs associated with the IQ block. See Table 21-92.

Table 21-92. CCR Names and Bit Positions

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1|2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INTO0	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INTO3	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INTO4	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INTO5	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily_DisplayModel signature of 0F_02. For processors with CPUID DisplayFamily_DisplayModel signature of 0F_00 and 0F_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINTOy bits is used.

Example 21-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR_CRU_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR_IQ_CCCR0. This enables CASCNT4INTO0 and OVF_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR_CRU_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR_IQ_CCCR4 (set ENABLE bit, but not OVF_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INTOy bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

21.6.3.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter overflows. The PMI interrupt service routine can then collect information about the state of the processor or program when overflow occurred. This information can then be used with a tool like the Intel® VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR_PMI flag in the counter's associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 21.6.3.5.6, "Cascading Counters." Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.

21.6.3.5.9 Counter Usage Guideline

There are some instances where the user must take care to configure counting logic properly, so that it is not powered down. To use any ESCR, even when it is being used just for tagging, (any) one of the counters that the particular ESCR (or its paired ESCR) can be connected to should be enabled. If this is not done, 0 counts may result. Likewise, to use any counter, there must be some event selected in a corresponding ESCR (other than no_event, which generally has a select value of 0).

21.6.3.6 At-Retirement Counting

At-retirement counting provides a means counting only events that represent work committed to architectural state and ignoring work that was performed speculatively and later discarded.

One example of this speculative activity is branch prediction. When a branch misprediction occurs, the results of instructions that were decoded and executed down the mispredicted path are canceled. If a performance counter was set up to count all executed instructions, the count would include instructions whose results were canceled as well as those whose results committed to architectural state.

To provide finer granularity in event counting in these situations, the performance monitoring facilities provided in the Pentium 4 and Intel Xeon processors provide a mechanism for tagging events and then counting only those tagged events that represent committed results. This mechanism is called "at-retirement counting."

There are predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire —** In at-retirement event descriptions, the term "bogus" refers to instructions or μops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or μops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and μops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, Instruction_Retired and Uops_Retired) can count instructions or μops that are retired based on the characterization of bogus" versus non-bogus.
- **Tagging —** Tagging is a means of marking μops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per μop and a direct count of the event would not provide an indication of how many μops encountered that event.

  The tagging mechanisms allow a μop to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per μop, rather than once per event. For example, a μop may encounter a cache miss more than once during its life time, but a "Miss Retired" metric (that counts the number of retired μops that encountered a cache miss) will increment only once for that μop. A "Miss Retired" metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the Intel® 64 and IA-32 Architectures Optimization Reference Manual (see Section 1.4, "Related Literature").
- **Replay —** To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules μops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied, μops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of μops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist —** When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of μops before they begin and are costly.

21.6.3.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and μops that encountered a specified event. For a subset of the at-retirement events, a μop may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count μops tagged with that mechanism:

- **Front-end tagging —** This mechanism pertains to the tagging of μops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the Front_end_event event.
- **Execution tagging —** This mechanism pertains to the tagging of μops that encountered execution events (for example, instruction types) and are counted with the Execution_Event event.

- **Replay tagging —** This mechanism pertains to tagging of μops whose retirement is replayed (for example, a cache miss) and are counted with the Replay_event event. Branch mispredictions are also tagged with this mechanism.
- **No tags —** This mechanism does not use tags. It uses the Instr_retired and the Uops_ retired events.

Each tagging mechanism is independent from all others; that is, a μop that has been tagged using one mechanism will not be detected with another mechanism's tagged-μop detector. For example, if μops are tagged using the front-end tagging mechanisms, the Replay_event will not count those as tagged μops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of μops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of μops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

There are performance monitoring events that support at-retirement counting: specifically the Front_end_event, Execution_event, Replay_event, Inst_retired, and Uops_retired events. The following sections describe the tagging mechanisms for using these events to tag μop and count tagged μops.

21.6.3.6.2 Tagging Mechanism for Front_end_event

The Front_end_event counts μops that have been tagged as encountering any of the following events:

- **μop decode events —** Tagging μops for μop decode events requires specifying bits in the ESCR associated with the performance-monitoring event, Uop_type.
- **Trace cache events —** Tagging μops for trace cache events may require specifying certain bits in the MSR_TC_PRECISE_EVENT MSR.

The MSRs that are supported by the front-end tagging mechanism must be set and one or both of the NBOGUS and BOGUS bits in the Front_end_event event mask must be set to count events. None of the events currently supported requires the use of the MSR_TC_PRECISE_EVENT MSR.

21.6.3.6.3 Tagging Mechanism For Execution_event

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* ESCR is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* ESCR is used to detect μops that have been tagged with that tag value identifier using Execution_event for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular μop. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of NBOGUS0 should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged μops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. The tag enable and tag value bits are irrelevant for the downstream ESCR used to select the Execution_event.

The four separate tag bits allow the user to simultaneously but distinctly count up to four execution events at retirement. (This applies for interrupt-based event sampling. There are additional restrictions for PEBS as noted in Section 21.6.3.8.3, "Setting Up the PEBS Buffer.") It is also possible to detect or count combinations of events by setting multiple tag value bits in the upstream ESCR or multiple mask bits in the downstream ESCR. For example, use a tag value of 3H in the upstream ESCR and use NBOGUS0/NBOGUS1 in the downstream ESCR event mask.

21.6.3.7 Tagging Mechanism for Replay_event

The replay mechanism enables tagging of μops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of μop that may experience the replay in the MSR_PEBS_MATRIX_VERT MSR and selecting the type of event in the MSR_PEBS_ENABLE MSR. Replay tagging must also be enabled with the UOP_Tag flag (bit 24) in the MSR_PEBS_ENABLE MSR.

The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 19.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA_32_PEBS_ENABLE_MSR. Each of these metrics requires that the Replay_Event be used to count the tagged μops.

21.6.3.8 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 19.4.5, "Branch Trace Store (BTS)," for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 19.4.9, "BTS and DS Save Area"). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and EFLAGS registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can only be carried out using the one performance counter, the MSR_IQ_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32_PMC0 and IA32_PERFEVTSEL0 MSRs (See Section 21.6.2.4).

21.6.3.8.1 Detection of the Availability of the PEBS Facilities

The DS feature flag (bit 21) returned by the CPUID instruction indicates (when set) the availability of the DS mechanism in the processor, which supports the PEBS (and BTS) facilities. When this bit is set, the following PEBS facilities are available:

- The PEBS_UNAVAILABLE flag in the IA32_MISC_ENABLE MSR indicates (when clear) the availability of the PEBS facilities, including the MSR_PEBS_ENABLE MSR.
- The enable PEBS flag (bit 24) in the MSR_PEBS_ENABLE MSR allows PEBS to be enabled (set) or disabled (clear).
- The IA32_DS_AREA MSR can be programmed to point to the DS save area.

21.6.3.8.2 Setting Up the DS Save Area

Section 19.4.9.2, "Setting Up the DS Save Area," describes how to set up and enable the DS save area. This procedure is common for PEBS and BTS.

21.6.3.8.3 Setting Up the PEBS Buffer

Only the MSR_IQ_COUNTER4 performance counter can be used for PEBS. Use the following procedure to set up the processor and this counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, and precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area (see Figure 19-5) to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS flag (bit 24) in MSR_PEBS_ENABLE MSR.
3. Set up the MSR_IQ_COUNTER4 performance counter and its associated CCCR and one or more ESCRs for PEBS.

21.6.3.8.4 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 19.4.9.5, “Writing the DS Interrupt Service Routine,” for guidelines for writing the DS ISR.

21.6.3.8.5 Other DS Mechanism Implications

The DS mechanism is not available in the SMM. It is disabled on transition to the SMM mode. Similarly the DS mechanism is disabled on the generation of a machine check exception and is cleared on processor RESET and INIT.

The DS mechanism is available in real address mode.

21.6.3.9 Operating System Implications

The DS mechanism can be used by the operating system as a debugging extension to facilitate failure analysis. When using this facility, a 25 to 30 times slowdown can be expected due to the effects of the trace store occurring on every taken branch.

Depending upon intended usage, the instruction pointers that are part of the branch records or the PEBS records need to have an association with the corresponding process. One solution requires the ability for the DS specific operating system module to be chained to the context switch. A separate buffer can then be maintained for each process of interest and the MSR pointing to the configuration area saved and setup appropriately on each context switch.

If the BTS facility has been enabled, then it must be disabled and state stored on transition of the system to a sleep state in which processor context is lost. The state must be restored on return from the sleep state.

It is required that an interrupt gate be used for the DS interrupt as opposed to a trap gate to prevent the generation of an endless interrupt loop.

Pages that contain buffers must have mappings to the same physical address for all processes/logical processors, such that any change to CR3 will not change DS addresses. If this requirement cannot be satisfied (that is, the feature is enabled on a per thread/process basis), then the operating system must ensure that the feature is enabled/disabled appropriately in the context switch code.

21.6.4 Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

The performance monitoring capability of processors based on Intel NetBurst microarchitecture and supporting Intel Hyper-Threading Technology is similar to that described in Section 21.6.3. However, the capability is extended so that:

- Performance counters can be programmed to select events qualified by logical processor IDs.
- Performance monitoring interrupts can be directed to a specific logical processor within the physical processor.

The sections below describe performance counters, event qualification by logical processor ID, and special purpose bits in ESCRs/CCCRs. They also describe MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_TC_PRECISE_EVENT.

21.6.4.1 ESCR MSRs

Figure 21-53 shows the layout of an ESCR MSR in processors supporting Intel Hyper-Threading Technology.

The functions of the flags and fields are as follows:

- **T1_USR flag, bit 0 —** When set, events are counted when thread 1 (logical processor 1) is executing at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.

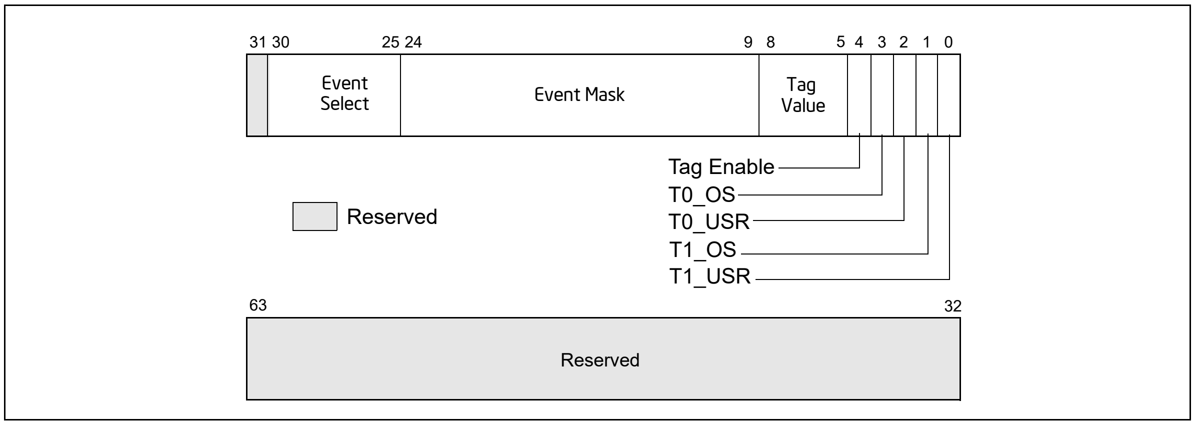


Figure 21-53. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel® Xeon® Processor, and Intel® Xeon® Processor MP Supporting Hyper-Threading Technology

- **T1_OS flag, bit 1 —** When set, events are counted when thread 1 (logical processor 1) is executing at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the T1_OS and T1_USR flags are set, thread 1 events are counted at all privilege levels.)
- **T0_USR flag, bit 2 —** When set, events are counted when thread 0 (logical processor 0) is executing at a CPL of 1, 2, or 3.
- **T0_OS flag, bit 3 —** When set, events are counted when thread 0 (logical processor 0) is executing at CPL of 0. (When both the T0_OS and T0_USR flags are set, thread 0 events are counted at all privilege levels.)
- **Tag enable, bit 4 —** When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 21.6.3.6, "At-Retirement Counting."
- **Tag value field, bits 5 through 8 —** Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0_OS and T0_USR flags and the T1_OS and T1_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 10.4.5, "Identifying Logical Processors in an MP System," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 21.6.4.4, "Performance Monitoring Events"). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

21.6.4.2 CCCR MSRs

Figure 21-54 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

- **Enable flag, bit 12 —** When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset
- **ESCR select field, bits 13 through 15 —** Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Active thread field, bits 16 and 17 —** Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:

  00 — None. Count only when neither logical processor is active.

01 — Single. Count only when one logical processor is active (either 0 or 1).

10 — Both. Count only when both logical processors are active.

11 — Any. Count when either logical processor is active.

A halted logical processor or a logical processor in the “wait for SIPI” state is considered inactive.

- **Compare flag, bit 18 —** When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.

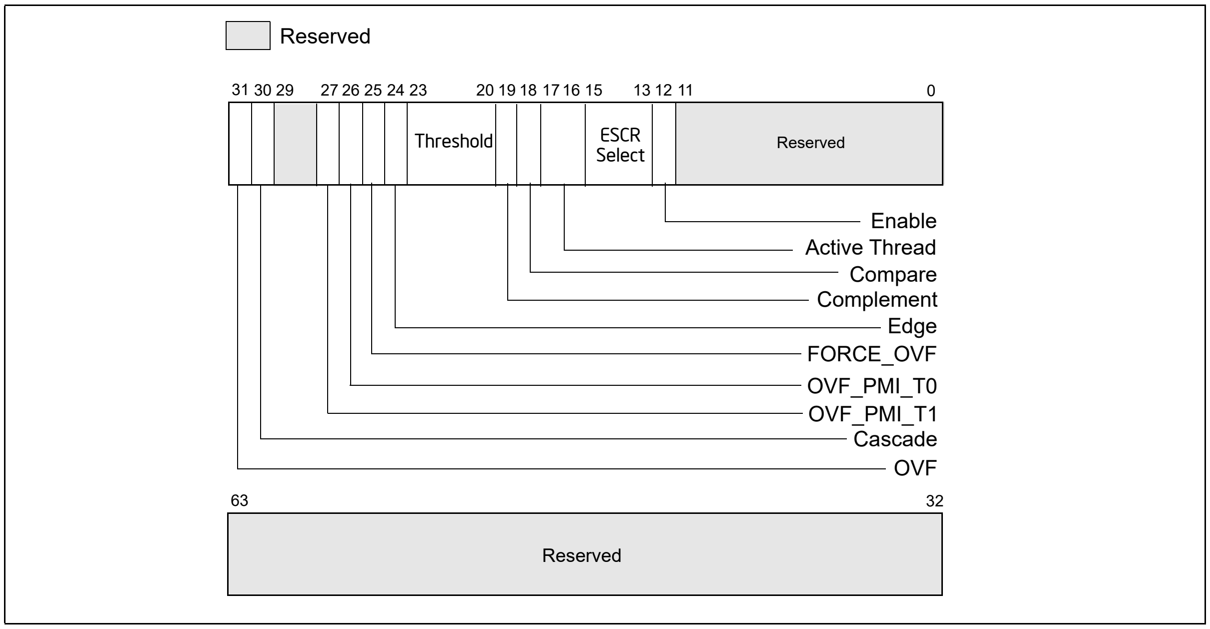


Figure 21-54. Counter Configuration Control Register (CCCR)

- **Complement flag, bit 19 —** Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 21.6.3.5.2, “Filtering Events”). The compare flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23 —** Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 21.6.3.5.2, “Filtering Events”).
- **Edge flag, bit 24 —** When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
- **FORCE_OVF flag, bit 25 —** When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI_T0 flag, bit 26 —** When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.
- **OVF_PMI_T1 flag, bit 27 —** When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.
- **Cascade flag, bit 30 —** When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 21.6.3.2, “Performance Counters,” for further details); when clear, disables cascading of counters.

- **OVF flag, bit 31 —** Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

21.6.4.3 IA32_PEBS_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR_PEBS_ENABLE MSR: bit 25 (ENABLE_PEBS_MY_THR) and 26 (ENABLE_PEBS_OTH_THR) respectively. These bits do not explicitly identify a specific logical processor by logic processor ID(T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running ("my thread") or for the other logical processor in the physical package on which the agent is not running ("other thread").

PEBS is supported for only a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can be carried out only with two performance counters: MSR_IQ_CCCR4 (MSR address 370H) for logical processor 0 and MSR_IQ_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE_PEBS_MY_THR and ENABLE_PEBS_OTH_THR bits in the MSR_PEBS_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

21.6.4.4 Performance Monitoring Events

When Intel Hyper-Threading Technology is active, many performance monitoring events can be can be qualified by the logical processor ID, which corresponds to bit 0 of the initial APIC ID. This allows for counting an event in any or all of the logical processors. However, not all the events have this logic processor specificity, or thread specificity.

Here, each event falls into one of two categories:

- **Thread specific (TS) —** The event can be qualified as occurring on a specific logical processor.
- **Thread independent (TI) —** The event cannot be qualified as being associated with a specific logical processor.

If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 21-93) depends only on the setting of the T0_USR and T0_OS flags in the ESCR being used to set up the event counter. The T1_USR and T1_OS flags have no effect on the count.

Table 21-93. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while T1 in USR	Counts while T1 in OS or USR	Counts while T1 in OS
T0_OS/T0_USR = 01	Counts while T0 in USR	Counts while T0 in USR or T1 in USR	Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 11	Counts while T0 in OS or USR	Counts while (a) T0 in OS or (b) T0 in USR or (c) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) or T0 in USR or (c) T1 in OS
T0_OS/T0_USR = 10	Counts T0 in OS	Counts T0 in OS or T1 in USR	Counts while (a)T0 in Os or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS

When a bit in the event mask field is TI, the effect of specifying bit-0-3 of the associated ESCR are described in Table 15-6. For events that are marked as TI, the effect of selectively specifying T0_USR, T0_OS, T1_USR, T1_OS bits is shown in Table 21-94.

Table 21-94. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 01	Counts while (a) T0 in USR or (b) T1 in USR	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 11	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 0	Counts while (a) T0 in OS or (b) T1 in OS	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS

21.6.4.5 Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

21.6.4.5.1 Non-Halted Clockticks

Use the following procedure to program ESCRs and CCCRs to obtain non-halted clockticks on processors based on Intel NetBurst microarchitecture:

1. Select an ESCR for the global_power_events and specify the RUNNING sub-event mask and the desired T0_OS/T0_USR/T1_OS/T1_USR bits for the targeted processor.
2. Select an appropriate counter.
3. Enable counting in the CCCR for that counter by setting the enable bit.

21.6.4.5.2 Non-Sleep Clockticks

Performance monitoring counters can be configured to count clockticks whenever the performance monitoring hardware is not powered-down. To count Non-sleep Clockticks with a performance-monitoring counter, do the following:

1. Select one of the 18 counters.
2. Select any of the ESCRs whose events the selected counter can count. Set its event select to anything other than "no_event"; the counter may be disabled if this is not done.
3. Turn threshold comparison on in the CCCR by setting the compare bit to "1".
4. Set the threshold to "15" and the complement to "1" in the CCCR. Since no event can exceed this threshold, the threshold condition is met every cycle and the counter counts every cycle. Note that this overrides any qualification (e.g., by CPL) specified in the ESCR.
5. Enable counting in the CCCR for the counter by setting the enable bit.

In most cases, the counts produced by the non-halted and non-sleep metrics are equivalent if the physical package supports one logical processor and is not placed in a power-saving state. Operating systems may execute an HLT instruction and place a physical processor in a power-saving state.

On processors that support Intel Hyper-Threading Technology (Intel HT Technology), each physical package can support two or more logical processors. Current implementation of Intel HT Technology provides two logical processors for each physical processor. While both logical processors can execute two threads simultaneously, one logical processor may halt to allow the other logical processor to execute without sharing execution resources between two logical processors.

Non-halted Clockticks can be set up to count the number of processor clock cycles for each logical processor whenever the logical processor is not halted (the count may include some portion of the clock cycles for that logical processor to complete a transition to a halted state). Physical processors that support Intel HT Technology enter into a power-saving state if all logical processors halt.

The Non-sleep Clockticks mechanism uses a filtering mechanism in CCCRs. The mechanism will continue to increment as long as one logical processor is not halted or in a power-saving state. Applications may cause a processor to enter into a power-saving state by using an OS service that transfers control to an OS's idle loop. The idle loop then may place the processor into a power-saving state after an implementation-dependent period if there is no work for the processor.

21.6.5 Performance Monitoring and Dual-Core Technology

The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

In the case of Pentium D processor, each logical processor is associated with dedicated resources for performance monitoring. In the case of Pentium processor Extreme edition, each processor core has dedicated resources, but two logical processors in the same core share performance monitoring resources (see Section 21.6.4, "Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture").

21.6.6 Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache

The 64-bit Intel Xeon processor MP with up to 8-MByte L3 cache has a CPUID signature of family [0FH], model [03H or 04H]. Performance monitoring capabilities available to Pentium 4 and Intel Xeon processors with the same values (see Section 21.1 and Section 21.6.4) apply to the 64-bit Intel Xeon processor MP with an L3 cache.

The level 3 cache is connected between the system bus and IOQ through additional control logic. See Figure 21-55.

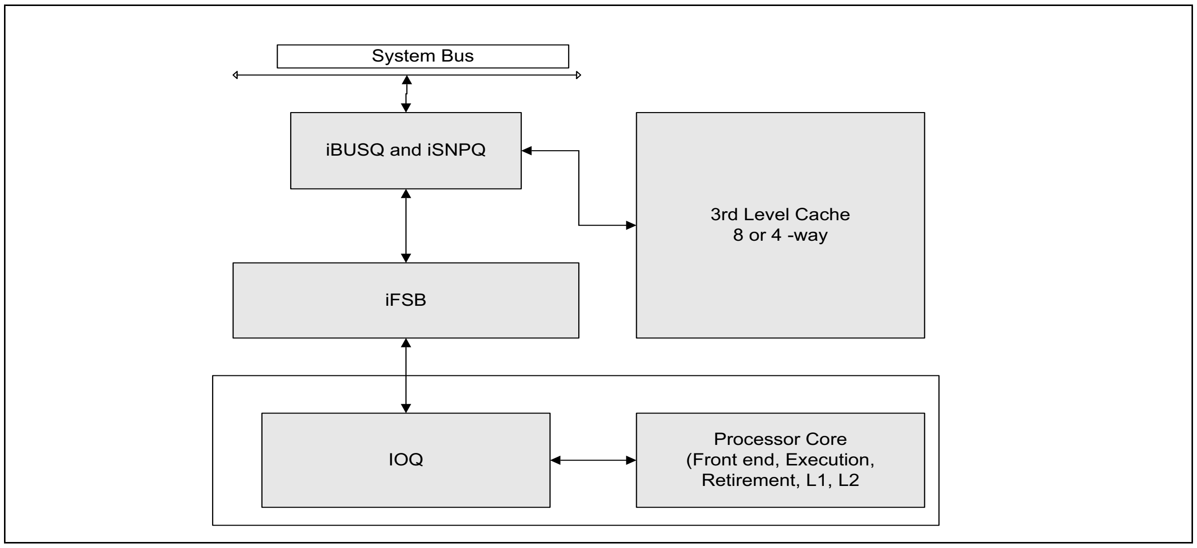


Figure 21-55. Block Diagram of 64-bit Intel® Xeon® Processor MP with 8-MByte L3

Additional performance monitoring capabilities and facilities unique to 64-bit Intel Xeon processor MP with an L3 cache are described in this section. The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs), each dedicated to a specific event. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values.

The lower 32-bits of the MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers. These performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:

- **IBUSQ event —** This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR_IFSB_IBUSQ0 and MSR_IFSB_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 21-56.

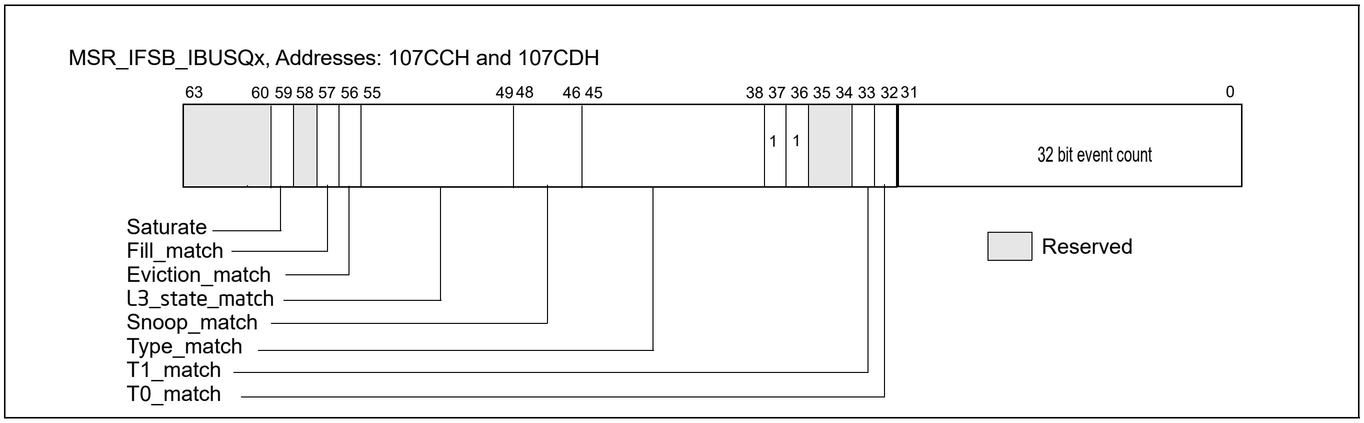


Figure 21-56. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH

- **ISNPQ event —** This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR_IFSB_ISNPQ0 and MSR_IFSB_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 21-57.

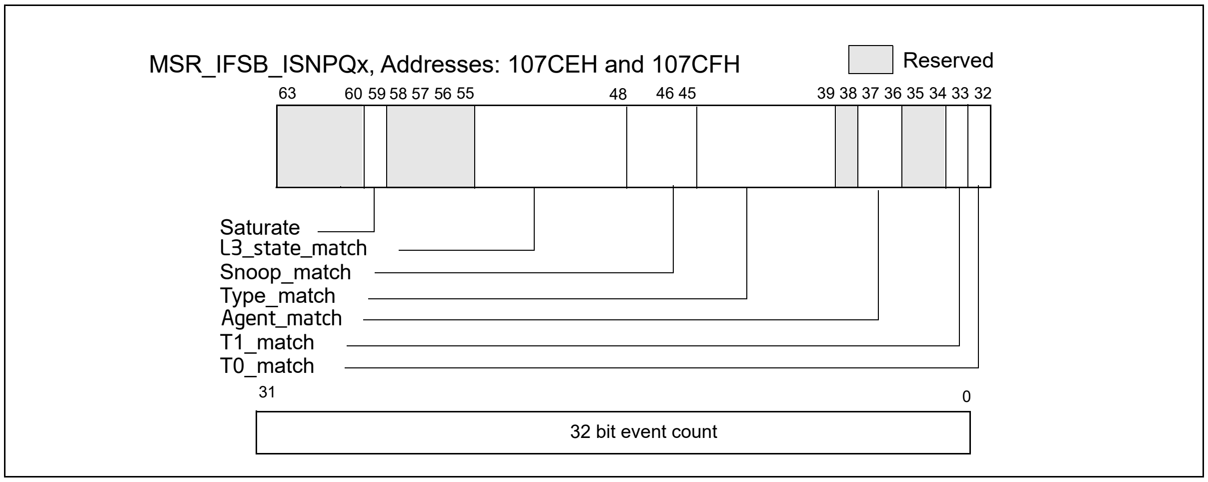


Figure 21-57. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH

- **EFSB event —** This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR_EFSB_DRDY0 and MSR_EFSB_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 21-58.

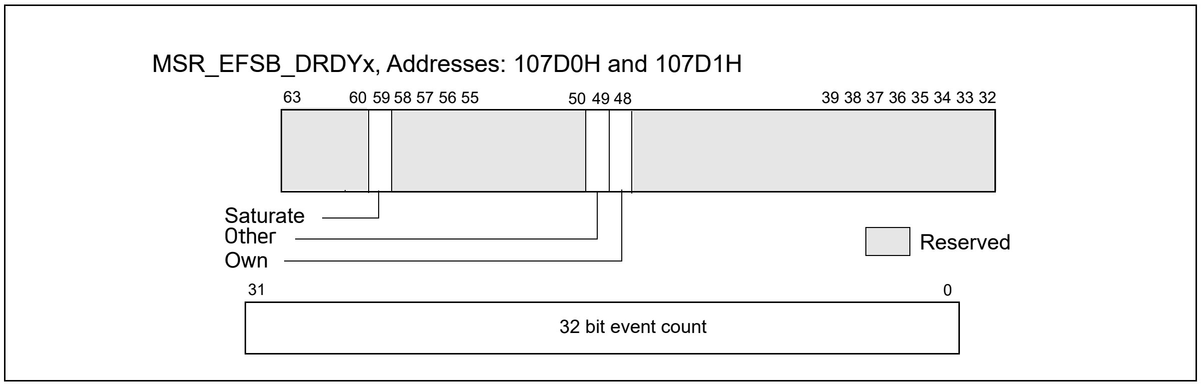


Figure 21-58. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H

- **IBUSQ Latency event —** This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR_IFSB_CTRL6[bit 26] to 1; the count freezes after software sets MSR_IFSB_CTRL6[bit 26] to 0. MSR_IFSB_CNTR7 acts as a 64-bit event counter for this event. See Figure 21-59.

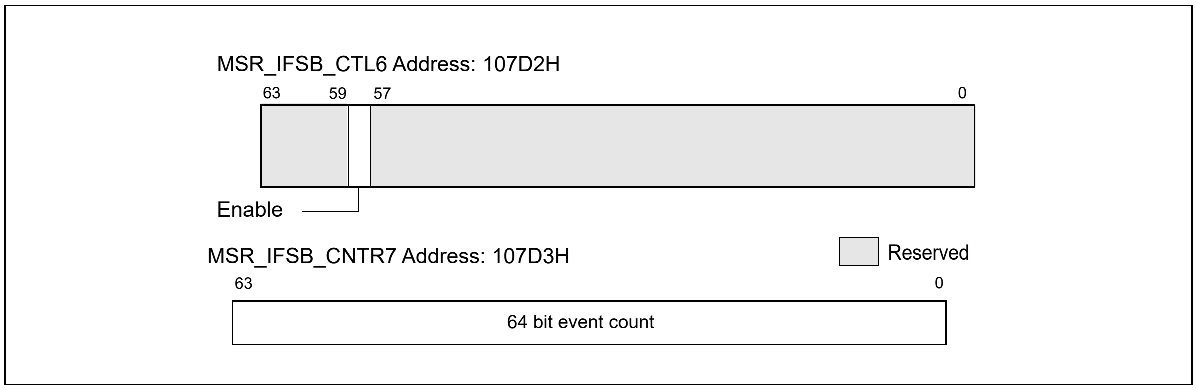


Figure 21-59. MSR_IFSB_CTL6, Address: 107D2H; MSR_IFSB_CNTR7, Address: 107D3H

21.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems

The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily_DisplayModel value of 06_1DH (see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through additional control logic. See Figure 21-60 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 21-60 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.

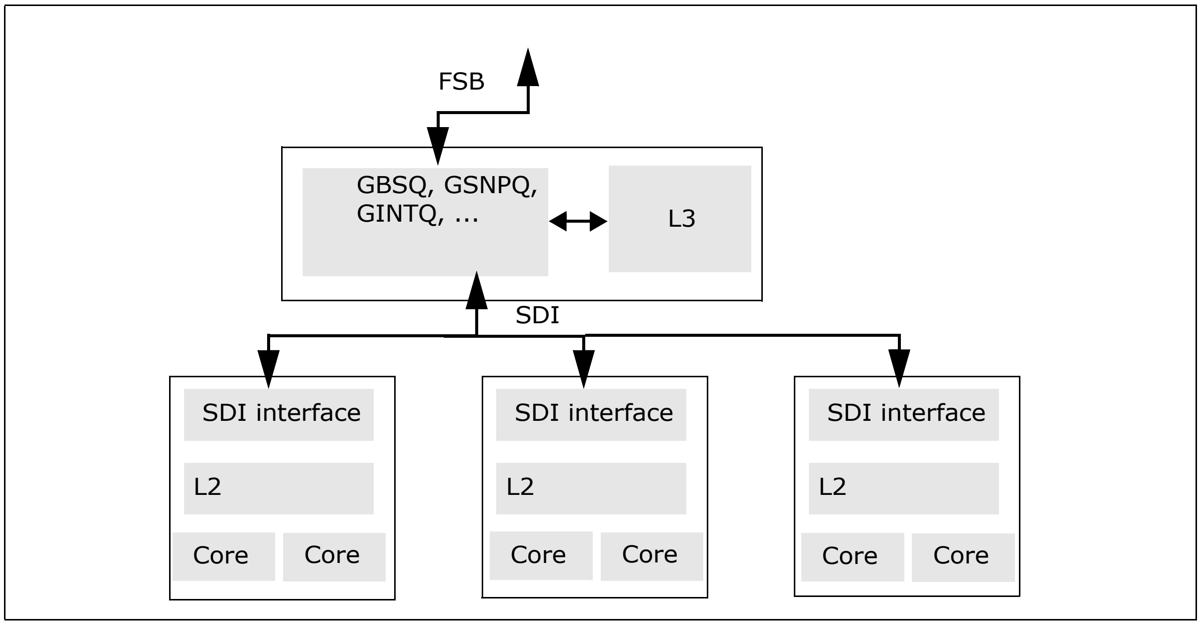


Figure 21-60. Block Diagram of the Intel® Xeon® Processor 7400 Series

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 21.1 and Section 21.6.4) apply to Intel Xeon processor 7100 series. The MSRs used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily_DisplayModel signature 06_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ_allocation and IOQ_active_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.

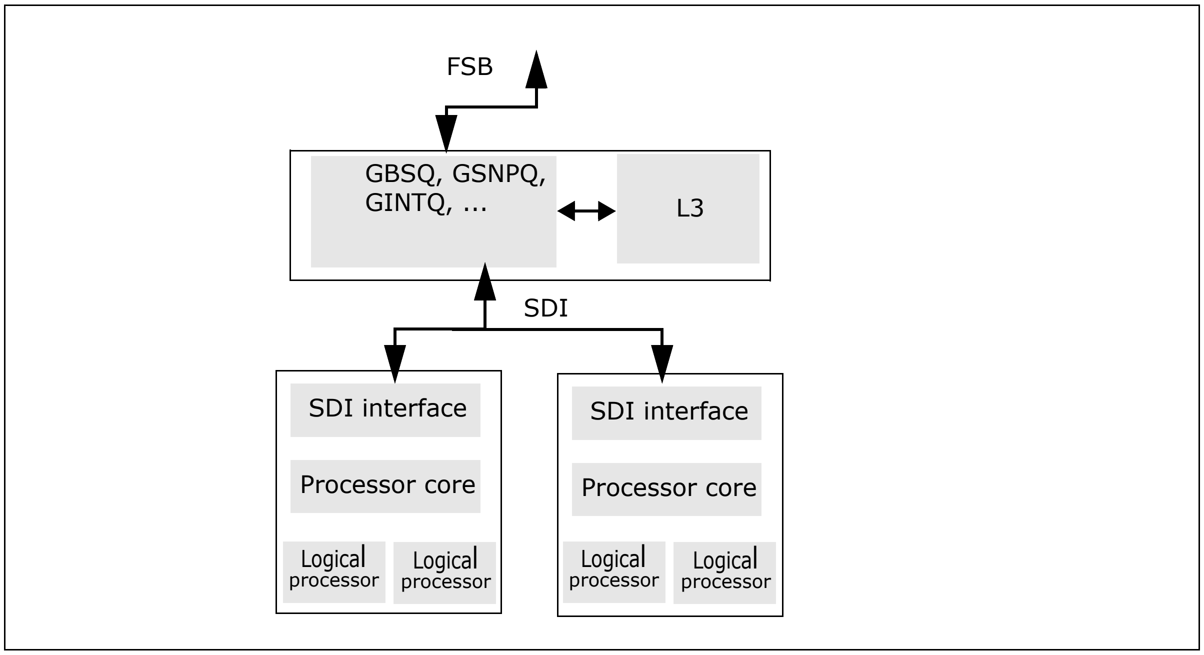


Figure 21-61. Block Diagram of the Intel® Xeon® Processor 7100 Series

21.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR_EMON_L3_GL_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3) are dedicated to counting GSNPQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Four MSRs (MSR_EMON_L3_CTR_CTL4, MSR_EMON_L3_CTR_CTL5, MSR_EMON_L3_CTR_CTL6, and MSR_EMON_L3_CTR_CTL7) are dedicated to counting external bus operations.

The bit fields in each of eight MSRs share the following common characteristics:

- Bits 63:32 is the event control field that includes an event mask and other bit fields that control counter operation. The event mask field specifies details of the microarchitectural condition, and its definition differs across GBSQ, GSNPQ, FSB.
- Bits 31:0 is the event count field. If the specified condition is met during each relevant clock domain of the event logic, the matched condition signals the counter logic to increment the associated event count field. The lower 32-bits of these 8 MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers.

In Dual-Core Intel Xeon processor 7100 series, the uncore performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

In Intel Xeon processor 7400 series, RDPMC with ECX between 2 and 9 can be used to access the eight uncore performance counter/control registers.

21.6.7.2 GBSQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1 is given in Figure 21-62. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following eight attributes:

- Agent_Select (bits 35:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, each bit specifies a logical processor in the physical package. The lower two bits corresponds to two logical processors in the first processor core, the upper two bits corresponds to two logical processors in the second processor core. 0FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each bit of [34:32] specifies the SDI logic of a dual-core module as the originator of the transaction. A value of 0111B in bits [35:32] specifies transaction from any processor core.

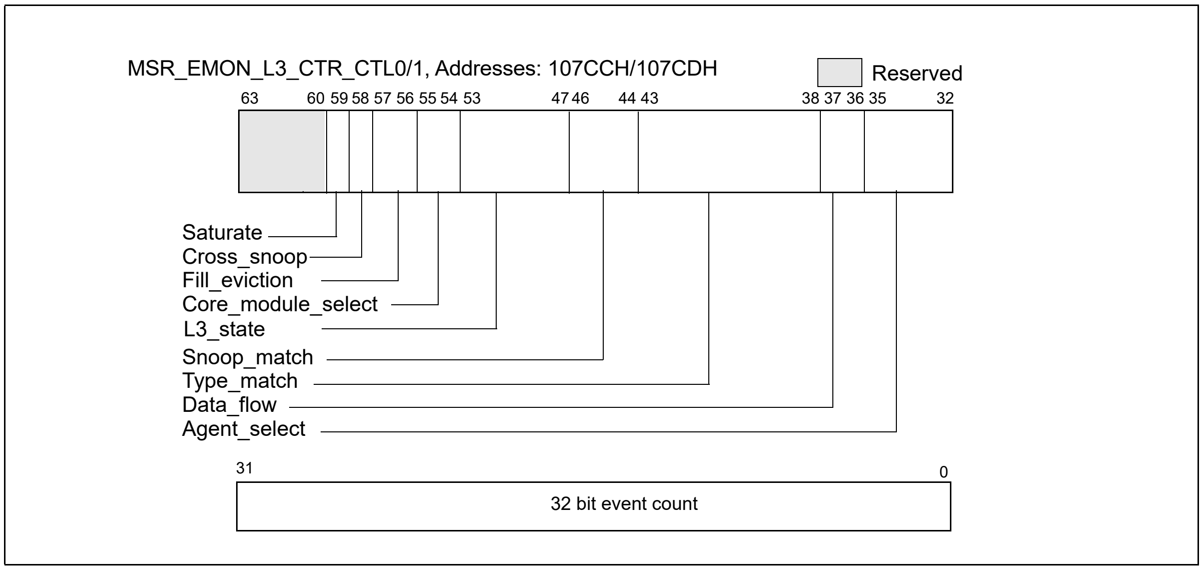


Figure 21-62. MSR_EMON_L3_CTR_CTL0/1, Addresses: 107CCH/107CDH

- Data_Flow (bits 37:36): Bit 36 specifies demand transactions, bit 37 specifies prefetch transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include all transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L3_State (bits 53:47): Each bit specifies an L2 coherency state.
- Core_Module_Select (bits 55:54): The valid encodings for L3 lookup differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series,

  — 00B: Match transactions from any core in the physical package

  — 01B: Match transactions from this core only

  — 10B: Match transactions from the other core in the physical package

  — 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series,

  — 00B: Match transactions from any dual-core module in the physical package

  — 01B: Match transactions from this dual-core module only

  — 10B: Match transactions from either one of the other two dual-core modules in the physical package

  - 11B: Match transaction from more than one dual-core modules in the physical package
- Fill_Eviction (bits 57:56): The valid encodings are
  - 00B: Match any transactions
  - 01B: Match transactions that fill L3
  - 10B: Match transactions that fill L3 without an eviction
  - 11B: Match transaction fill L3 with an eviction
- Cross_Snoop (bit 58): The encodings are
  - 0B: Match any transactions
  - 1B: Match cross snoop transactions

For each counting clock domain, if all eight attributes match, event logic signals to increment the event count field.

21.6.7.3 GSNPQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3 is given in Figure 21-63. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following six attributes:

- Agent_Select (bits 37:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
- For Intel Xeon processor 7100 series, each of the lowest 4 bits specifies a logical processor in the physical package. The lowest two bits corresponds to two logical processors in the first processor core, the next two bits corresponds to two logical processors in the second processor core. Bit 36 specifies other symmetric agent transactions. Bit 37 specifies central agent transactions. 3FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each of the lowest 3 bits specifies a dual-core module in the physical package. Bit 37 specifies central agent transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include any transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L2_State (bits 53:47): Each bit specifies an L3 coherency state.
- Core_Module_Select (bits 56:54): Bit 56 enables Core_Module_Select matching. If bit 56 is clear, Core_Module_Select encoding is ignored. The valid encodings for the lower two bits (bit 55, 54) differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one core (irrespective which core) in the physical package
  - 01B: Match transactions from this core and not the other core
  - 10B: Match transactions from the other core in the physical package, but not this core
  - 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one dual-core module (irrespective which module) in the physical package.
  - 01B: Match transactions from one or more dual-core modules.
  - 10B: Match transactions from two or more dual-core modules.
  - 11B: Match transaction from all three dual-core modules in the physical package.
- Block_Snoop (bit 57): specifies blocked snoop.

For each counting clock domain, if all six attributes match, event logic signals to increment the event count field.

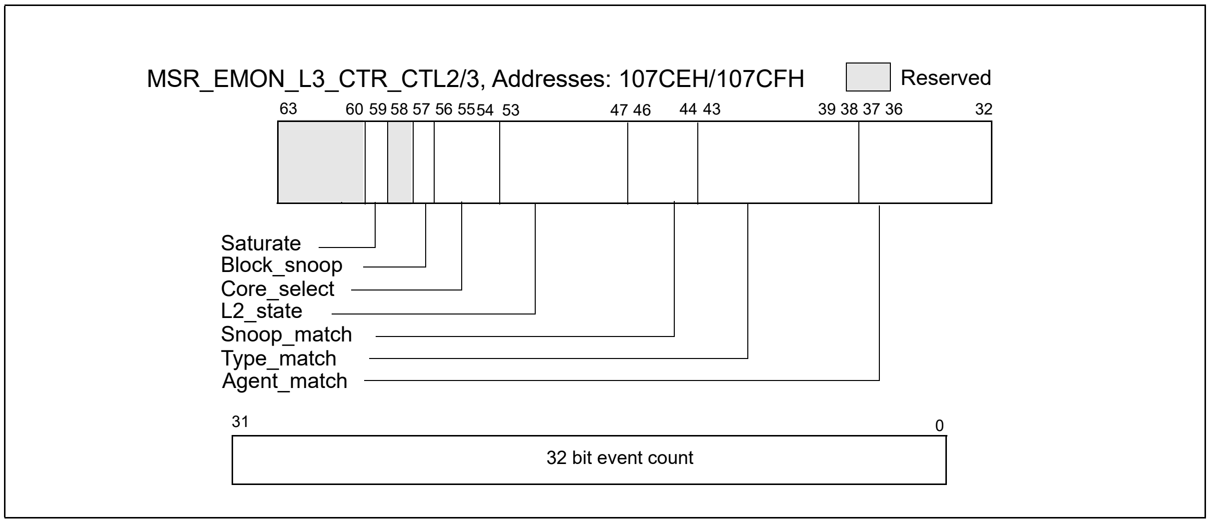


Figure 21-63. MSR_EMON_L3_CTR_CTL2/3, Addresses: 107CEH/107CFH

21.6.7.4 FSB Event Interface

The layout of MSR_EMON_L3_CTR_CTL4 through MSR_EMON_L3_CTR_CTL7 is given in Figure 21-64. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) is organized as follows:

- Bit 58: must set to 1.
- FSB_Submask (bits 57:32): Specifies FSB-specific sub-event mask.

The FSB sub-event mask defines a set of independent attributes. The event logic signals to increment the associated event count field if one of the attribute matches. Some of the sub-event mask bit counts durations. A duration event increments at most once per cycle.

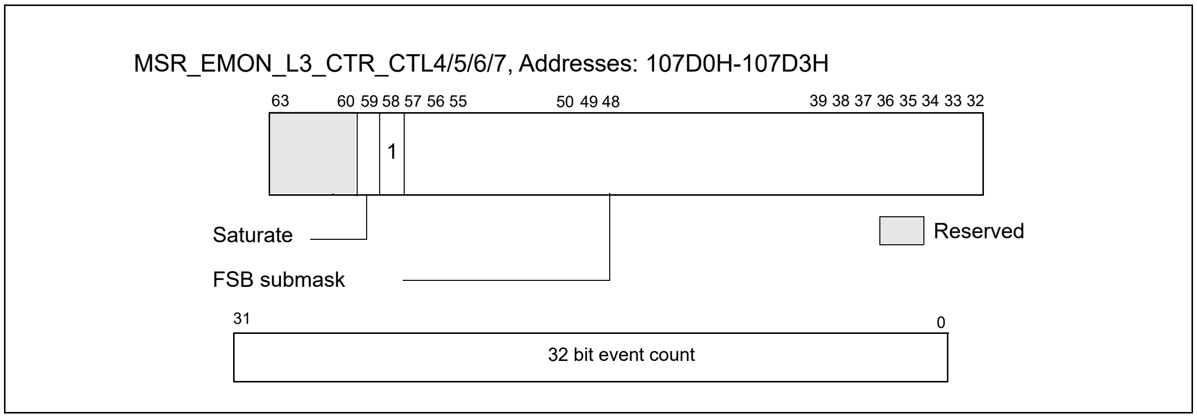


Figure 21-64. MSR_EMON_L3_CTR_CTL4/5/6/7, Addresses: 107D0H-107D3H

21.6.7.4.1 FSB Sub-Event Mask Interface

- FSB_type (bit 37:32): Specifies different FSB transaction types originated from this physical package.
- FSB_L_clear (bit 38): Count clean snoop results from any source for transaction originated from this physical package.
- FSB_L_hit (bit 39): Count HIT snoop results from any source for transaction originated from this physical package.

- FSB_L_hitm (bit 40): Count HITM snoop results from any source for transaction originated from this physical package.
- FSB_L_defer (bit 41): Count DEFER responses to this processor's transactions.
- FSB_L_retry (bit 42): Count RETRY responses to this processor's transactions.
- FSB_L_snoop_stall (bit 43): Count snoop stalls to this processor's transactions.
- FSB_DBSY (bit 44): Count DBSY assertions by this processor (without a concurrent DRDY).
- FSB_DRDY (bit 45): Count DRDY assertions by this processor.
- FSB_BNR (bit 46): Count BNR assertions by this processor.
- FSB_IOQ_empty (bit 47): Counts each bus clocks when the IOQ is empty.
- FSB_IOQ_full (bit 48): Counts each bus clocks when the IOQ is full.
- FSB_IOQ_active (bit 49): Counts each bus clocks when there is at least one entry in the IOQ.
- FSB_WW_data (bit 50): Counts back-to-back write transaction's data phase.
- FSB_WW_issue (bit 51): Counts back-to-back write transaction request pairs issued by this processor.
- FSB_WR_issue (bit 52): Counts back-to-back write-read transaction request pairs issued by this processor.
- FSB_RW_issue (bit 53): Counts back-to-back read-write transaction request pairs issued by this processor.
- FSB_other_DBSY (bit 54): Count DBSY assertions by another agent (without a concurrent DRDY).
- FSB_other_DRDY (bit 55): Count DRDY assertions by another agent.
- FSB_other_snoop_stall (bit 56): Count snoop stalls on the FSB due to another agent.
- FSB_other_BNR (bit 57): Count BNR assertions from another agent.

21.6.7.5 Common Event Control Interface

The MSR_EMON_L3_GL_CTL MSR provides simplified access to query overflow status of the GBSQ, GSNPQ, FSB event counters. It also provides control bit fields to freeze, unfreeze, or reset those counters. The following bit fields are supported:

- GL_freeze_cmd (bit 0): Freeze the event counters specified by the GL_event_select field.
- GL_unfreeze_cmd (bit 1): Unfreeze the event counters specified by the GL_event_select field.
- GL_reset_cmd (bit 2): Clear the event count field of the event counters specified by the GL_event_select field. The event select field is not affected.
- GL_event_select (bit 23:16): Selects one or more event counters to subject to specified command operations indicated by bits 2:0. Bit 16 corresponds to MSR_EMON_L3_CTR_CTL0, bit 23 corresponds to MSR_EMON_L3_CTR_CTL7.
- GL_event_status (bit 55:48): Indicates the overflow status of each event counters. Bit 48 corresponds to MSR_EMON_L3_CTR_CTL0, bit 55 corresponds to MSR_EMON_L3_CTR_CTL7.

In the event control field (bits 63:32) of each MSR, if the saturate control (bit 59, see Figure 21-62 for example) is set, the event logic forces the value FFFF_FFFFH into the event count field instead of incrementing it.

21.6.8 Performance Monitoring (P6 Family Processor)

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These can either count events or measure duration. When counting events, a counter increments each time a specified event takes place or a specified number of events takes place. When measuring duration, it counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.

NOTE

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The performance-monitoring counters are supported by four MSRs: the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

NOTE

The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed for P6 family processors are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

21.6.8.1 PerfEvtSel0 and PerfEvtSel1 MSRs

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Figure 21-65 shows the flags and fields in these MSRs.

The functions of the flags and fields in the PerfEvtSel0 and PerfEvtSel1 MSRs are as follows:

- **Event select field (bits 0 through 7) —** Selects the event logic unit to detect certain microarchitectural conditions.
- **Unit mask (UMASK) field (bits 8 through 15) —** Further qualifies the event logic unit selected in the event select field to detect a specific microarchitectural condition. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states.

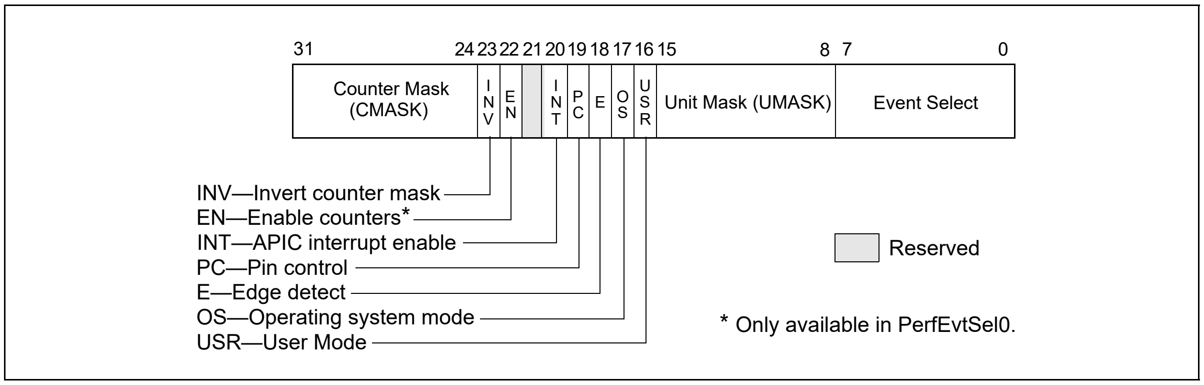


Figure 21-65. PerfEvtSel0 and PerfEvtSel1 MSRs

- **USR (user mode) flag (bit 16) —** Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.
- **OS (operating system mode) flag (bit 17) —** Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.
- **E (edge detect) flag (bit 18) —** Enables (when set) edge detection of events. The processor counts the number of deasserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19) —** When set, the processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20) —** When set, the processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31) —** When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

21.6.8.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value, and the high-order 8 bits are sign-extended according to the value of bit 31. This operation allows writing both positive and negative values to the performance counters.

21.6.8.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

21.6.8.4 Event and Time-Stamp Monitoring Software

To use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking.
- Initialize and start counters.
- Stop counters.
- Read the event counters.
- Read the time-stamp counter.

The event monitor feature determination procedure must check whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. (The Pentium and P6 family processors support performance counters.) The procedure also checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialize and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be counted and the method used to count them and initializes the counter MSRs (PerfCtr0 and PerfCtr1) to starting counts. The stop counters procedure stops the performance counters (see Section 21.6.8.3, "Starting and Stopping the Performance-Monitoring Counters").

The read counters procedure reads the values in the PerfCtr0 and PerfCtr1 MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures would be provided in lieu of enabling the RDTSC and RDPMC instructions that allow application code to read the counters.

21.6.8.5 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

21.6.9 Performance Monitoring (Pentium Processors)

The Pentium processor provides two 40-bit performance counters, which can be used to count events or measure duration. The counters are supported by three MSRs: the control and event select MSR (CESR) and the performance counter MSRs (CTR0 and CTR1). These can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0.

Each counter has an associated external pin (PM0/BP0 and PM1/BP1), which can be used to indicate the state of the counter to external hardware.

NOTES

The CESR, CTR0, and CTR1 MSRs and the events listed for Pentium processors are model-specific for the Pentium processor.

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

21.6.9.1 Control and Event Select Register (CESR)

The 32-bit control and event select MSR (CESR) controls the operation of performance-monitoring counters CTR0 and CTR1 and the associated pins (see Figure 21-66). To control each counter, the CESR register contains a 6-bit event select field (ES0 and ES1), a pin control flag (PC0 and PC1), and a 3-bit counter control field (CC0 and CC1). The functions of these fields are as follows:

- **ES0 and ES1 (event select) fields (bits 0-5, bits 16-21) —** Selects (by entering an event code in the field) up to two events to be monitored.

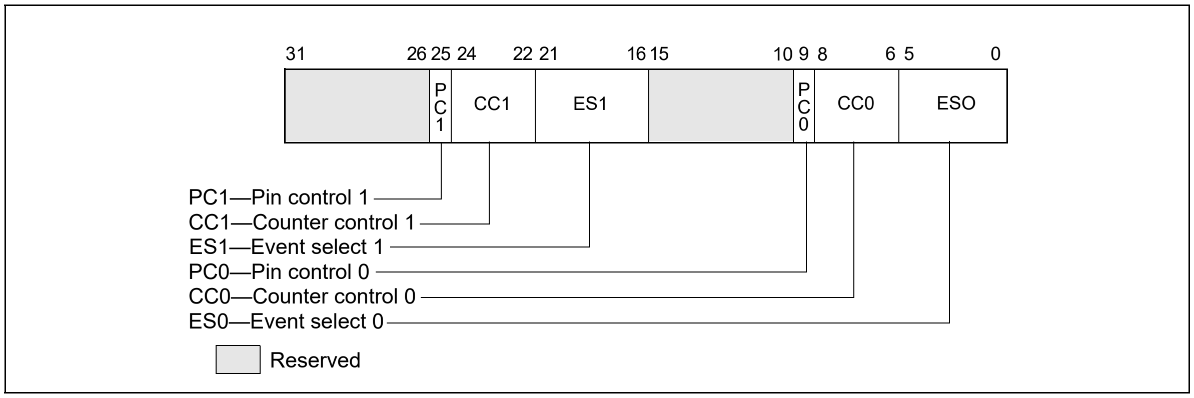


Figure 21-66. CESR MSR (Pentium Processor Only)

- **CC0 and CC1 (counter control) fields (bits 6-8, bits 22-24) —** Controls the operation of the counter. Control codes are as follows:

  000 — Count nothing (counter disabled).

  001 — Count the selected event while CPL is 0, 1, or 2.

  010 — Count the selected event while CPL is 3.

  011 — Count the selected event regardless of CPL.

  100 — Count nothing (counter disabled).

  101 — Count clocks (duration) while CPL is 0, 1, or 2.

  110 — Count clocks (duration) while CPL is 3.

  111 — Count clocks (duration) regardless of CPL.

  The highest order bit selects between counting events and counting clocks (duration); the middle bit enables counting when the CPL is 3; and the low-order bit enables counting when the CPL is 0, 1, or 2.

- **PC0 and PC1 (pin control) flags (bits 9, 25) —** Selects the function of the external performance-monitoring counter pin (PM0/BP0 and PM1/BP1). Setting one of these flags to 1 causes the processor to assert its associated pin when the counter has overflowed; setting the flag to 0 causes the pin to be asserted when the counter has been incremented. These flags permit the pins to be individually programmed to indicate the overflow or incremented condition. The external signaling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

21.6.9.2 Use of the Performance-Monitoring Pins

When performance-monitor pins PM0/BP0 and/or PM1/BP1 are configured to indicate when the performance-monitor counter has incremented and an "occurrence event" is being counted, the associated pin is asserted (high) each time the event occurs. When a "duration event" is being counted, the associated PM pin is asserted for the entire duration of the event. When the performance-monitor pins are configured to indicate when the counter has overflowed, the associated PM pin is asserted when the counter has overflowed.

When the PM0/BP0 and/or PM1/BP1 pins are configured to signal that a counter has incremented, it should be noted that although the counters may increment by 1 or 2 in a single clock, the pins can only indicate that the event occurred. Moreover, since the internal clock frequency may be higher than the external clock frequency, a single external clock may correspond to multiple internal clocks.

A "count up to" function may be provided when the event pin is programmed to signal an overflow of the counter. Because the counters are 40 bits, a carry out of bit 39 indicates an overflow. A counter may be preset to a specific value less then $2^{40} – 1$. After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

21.6.9.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).
- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

21.7 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.
- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor's bus clock frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions.
- Enhanced Intel SpeedStep Technology transitions (P-state transitions).

For Intel processors that support TM2, the processor core clocks may operate at a frequency that differs from the Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See Section 21.7.2 for more detail.

Due to the above considerations there are several important clocks referenced in this manual:

- **Base Clock —** The frequency of this clock is the frequency of the processor when the processor is not in turbo mode, and not being throttled via Intel SpeedStep.
- **Maximum Clock —** This is the maximum frequency of the processor when turbo mode is at the highest point.
- **Bus Clock —** These clockticks increment at a fixed frequency and help coordinate the bus on some systems.

- **Core Crystal Clock —** This is a clock that runs at fixed frequency; it coordinates the clocks on all packages across the system.
- **Non-halted Clockticks —** Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.
- **Non-sleep Clockticks —** Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.
- **Time-stamp Counter —** See Section 19.17, "Time-Stamp Counter."
- **Reference Clockticks —** TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 19.17, "Time-Stamp Counter," for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI —** Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.
- **Nominal CPI —** Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

21.7.1 Non-Halted Reference Clockticks

Software can use UnHalted Reference Cycles on either a general purpose performance counter using event mask 0x3C and UMASK 0x01 or on fixed function performance counter 2 to count at a constant rate. These events count at a consistent rate irrespective of P-state, TM2, or frequency transitions that may occur to the processor. The UnHalted Reference Cycles event may count differently on the general purpose event and fixed counter.

21.7.2 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 16, "Power and Thermal Management"), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

- The time stamp counter operates at a fixed-rate frequency of the processor.
- The IA32_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.
- The IA32_FIXED_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.
- The Local APIC timer operation is unaffected by opportunistic processor operation.
- The TSC, IA32_MPERF, and IA32_FIXED_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.

21.7.3 Determining the Processor Base Frequency

For Intel processors in which the nominal core crystal clock frequency is enumerated in CPUID.15H.ECX and the core crystal clock ratio is encoded in CPUID.15H (see Table 3-17 "Information Returned by CPUID Instruction"), the nominal TSC frequency can be determined by using the following equation:

Nominal TSC frequency = (CPUID.15H.ECX[31:0] * CPUID.15H.EBX[31:0]) ÷ CPUID.15H.EAX[31:0]

For Intel processors in which CPUID.15H.EBX[31:0] ÷ CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 21-95 can be used to look up the nominal core crystal clock frequency.

Table 21-95. Nominal Core Crystal Clock Frequency

Processor Families/Processor Number Series[1]	Nominal Core Crystal Clock Frequency
Intel® Xeon® Scalable Processor Family with CPUID signature 06_55H.	25 MHz
6th and 7th generation Intel® Core™ processors and Intel® Xeon® W Processor Family.	24 MHz
Next Generation Intel Atom® processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors).	19.2 MHz

NOTES:

1. For any processor in which CPUID.15H is enumerated and MSR_PLATFORM_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

21.7.3.1 For Intel® Processors Based on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell Microarchitectures

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 100 MHz.

21.7.3.2 For Intel® Processors Based on Nehalem Microarchitecture

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 133.33 MHz.

21.7.3.3 For Intel Atom® Processors Based on Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by the scalable bus frequency. The scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] for Intel Atom processors based on the Silvermont microarchitecture, and in bit field MSR_FSB_FREQ[3:0] for processors based on the Airmont microarchitecture; see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

21.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] at (0CDH), see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR_PLATFORM_ID[12:8]. It corresponds to the Processor Base frequency.

- IF XE operation is enabled, the maximum resolved bus ratio is given in MSR_PERF_STATUS[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR_PERF_STATUS[31] is set, XE operation is enabled. The MSR_PERF_STATUS[31] field is read-only.

21.8 IA32_PERF_CAPABILITIES MSR ENUMERATION

The layout of IA32_PERF_CAPABILITIES MSR is shown in Figure 21-67; it provides enumeration of a variety of interfaces:

- IA32_PERF_CAPABILITIES.LBR_FMT[bits 5:0]: encodes the LBR format, details are described in Section 19.4.8.1.
- IA32_PERF_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist; see Section 21.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers; see Section 21.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBS_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records; see Section 21.6.2.4.2.
- IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[12]: Indicates IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if 1. See Section 21.8.1.
- IA32_PERF_CAPABILITIES.FULL_WRITE[13]: Indicates the processor supports IA32_A_PMCx interface for updating bits 32 and above of IA32_PMCx; see Section 21.2.8.
- IA32_PERF_CAPABILITIES.PEBS_BASELINE [bit 14]: If set, the following is true:
  - The IA32_PEBS_ENABLE MSR (address 3F1H) exists and all architecturally enumerated fixed and general-purpose counters have corresponding bits in IA32_PEBS_ENABLE that enable generation of PEBS records. The general-purpose counter bits start at bit IA32_PEBS_ENABLE[0], and the fixed counter bits start at bit IA32_PEBS_ENABLE[32].
  - The format of the PEBS record is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT; see Section 21.6.2.4.2.
  - Extended PEBS is supported. All counters support the PEBS facility, and all events (both precise and non-precise) can generate PEBS records when PEBS is enabled for that counter. Note that not all events may be available on all counters.
  - Adaptive PEBS is supported. The PEBS_DATA_CFG MSR (address 3F2H) and adaptive record enable bits (IA32_PERFEVTSELx.Adaptive_Record and IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record) are supported. The definition of the PEBS_DATA_CFG MSR, including which bits are supported and how they affect the record, is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT. See Section 21.9.2.3.
  - NOTE: Software is recommended to feature PEBS Baseline when the following is true: IA32_PERF_CAPABILITIES.PEBS_BASELINE[14] && IA32_PERF_CAPABILITIES.PEBS_FMT[11:8] $\geq$ 4.
- IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE[15]: If set, indicates that the architecture provides built in support for TMA L1 metrics through the PERF_METRICS MSR. See Section 21.3.9.3.
- IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16]: If set on parts that enumerate support for Intel PT (CPUID.0x7.0.EBX[25]=1), setting IA32_PEBS_ENABLE.PEBS_OUTPUT to 01B will result in PEBS output being written into the Intel PT trace stream. See Section 21.5.5.2.
- IA32_PERF_CAPABILITIES.PEBS_TIMING_INFO[17]: If set, indicates that the processor supports the Timed PEBS capability. See Section 21.9.9.
- IA32_PERF_CAPABILITIES.RDPMC_METRICS_CLEAR[19]: If set, indicates that the processor supports RDPMC Metrics Clear Mode.

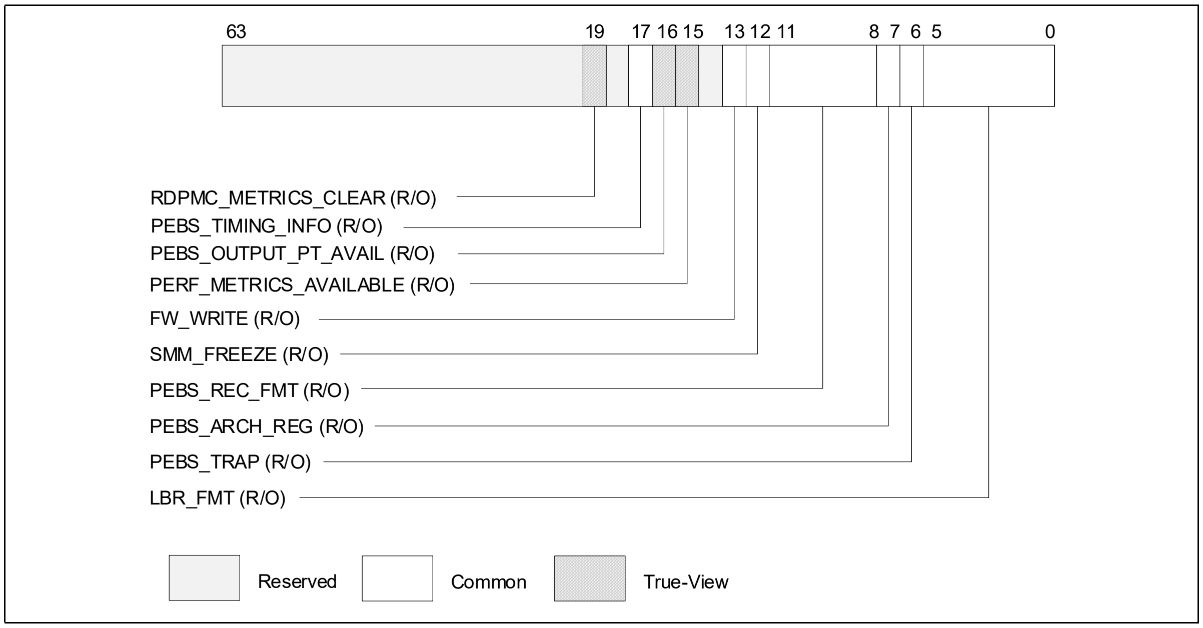


Figure 21-67. Layout of IA32_PERF_CAPABILITIES MSR

21.8.1 Filtering of SMM Handler Overhead

When performance monitoring facilities and/or branch profiling facilities (see Section 19.5, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel Atom® Processors)") are enabled, these facilities capture event counts, branch records and branch trace messages occurring in a logical processor. The occurrence of interrupts, instruction streams due to various interrupt handlers all contribute to the results recorded by these facilities.

If CPUID.01H:ECX.PDCM[bit 15] is 1, the processor supports the IA32_PERF_CAPABILITIES MSR. If IA32_PERF_-CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is 1, the processor supports the ability for system software using performance monitoring and/or branch profiling facilities to filter out the effects of servicing system management interrupts.

If the FREEZE_WHILE_SMM capability is enabled on a logical processor and after an SMI is delivered, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler.

The enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored , after the SMI handler issues RSM to complete its servicing.

It is the responsibility of the SMM code to ensure the state of the performance monitoring and branch profiling facilities are preserved upon entry or until prior to exiting the SMM. If any of this state is modified due to actions by the SMM code, the SMM code is required to restore such state to the values present at entry to the SMM handler.

System software is allowed to set IA32_DEBUGCTL.FREEZE_WHILE_SMM[bit 14] to 1 only supported as indicated by IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] reporting 1.

21.9 PEBS FACILITY

21.9.1 Extended PEBS

The Extended PEBS feature supports Processor Event Based Sampling (PEBS) on all counters, both fixed function and general purpose; and all performance monitoring events, both precise and non-precise. PEBS can be enabled for the general purpose counters using PEBS_EN_PMCi bits of IA32_PEBS_ENABLE (i = 0, 1,..m). PEBS can be enabled for 'i' fixed function counters using the PEBS_EN_FIXEDi bits of IA32_PEBS_ENABLE (i = 0, 1, ...n).

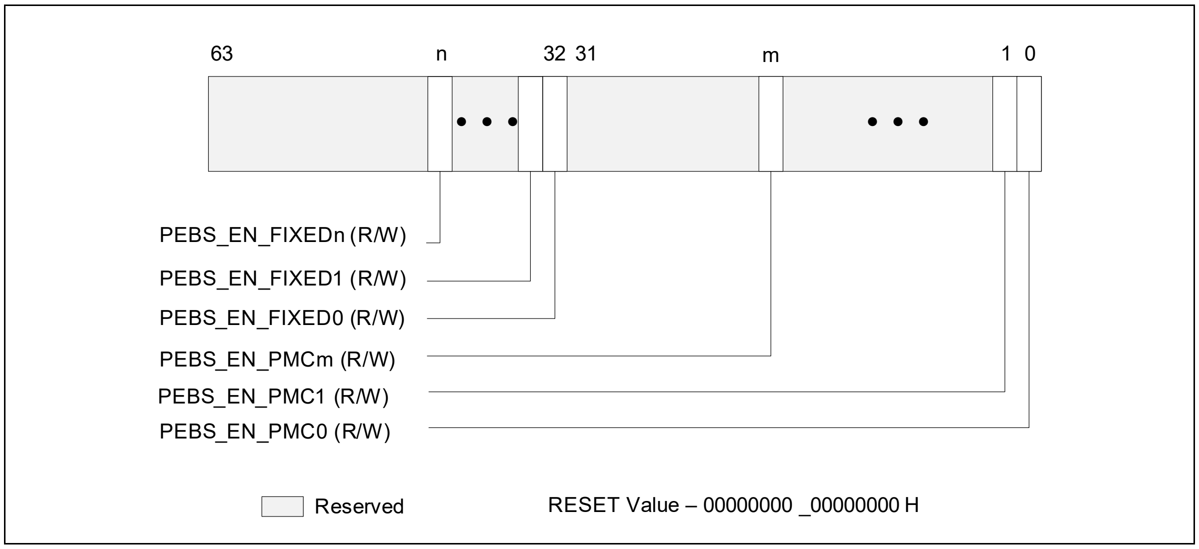


Figure 21-68. Layout of IA32_PEBS_ENABLE MSR

A PEBS record due to a precise event will be generated after an instruction that causes the event when the counter has already overflowed. A PEBS record due to a non-precise event will occur at the next opportunity after the counter has overflowed, including immediately after an overflow is set by an MSR write.

Currently, IA32_FIXED_CTR0 counts instructions retired and is a precise event. IA32_FIXED_CTR1, IA32_-FIXED_CTR2 … IA32_FIXED_CTRm count as non-precise events.

The Applicable Counter field in the Basic Info Group of the PEBS record indicates which counters caused the PEBS record to be generated. It is in the same format as the enable bits for each counter in IA32_PEBS_ENABLE. As an example, an Applicable Counter field with bits 2 and 32 set would indicate that both general purpose counter 2 and fixed function counter 0 generated the PEBS record.

To properly use PEBS for the additional counters, software will need to set up the counter reset values in PEBS portion of the DS_BUFFER_MANAGEMENT_AREA data structure that is indicated by the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 21-69. When a counter generates a PEBS records, the appropriate counter reset values will be loaded into that counter. In the above example where general purpose counter 2 and fixed function counter 0 generated the PEBS record, general purpose counter 2 would be reloaded with the value contained in PEBS GP Counter 2 Reset (offset 50H) and fixed function counter 0 would be reloaded with the value contained in PEBS Fixed Counter 0 Reset (offset 80H).

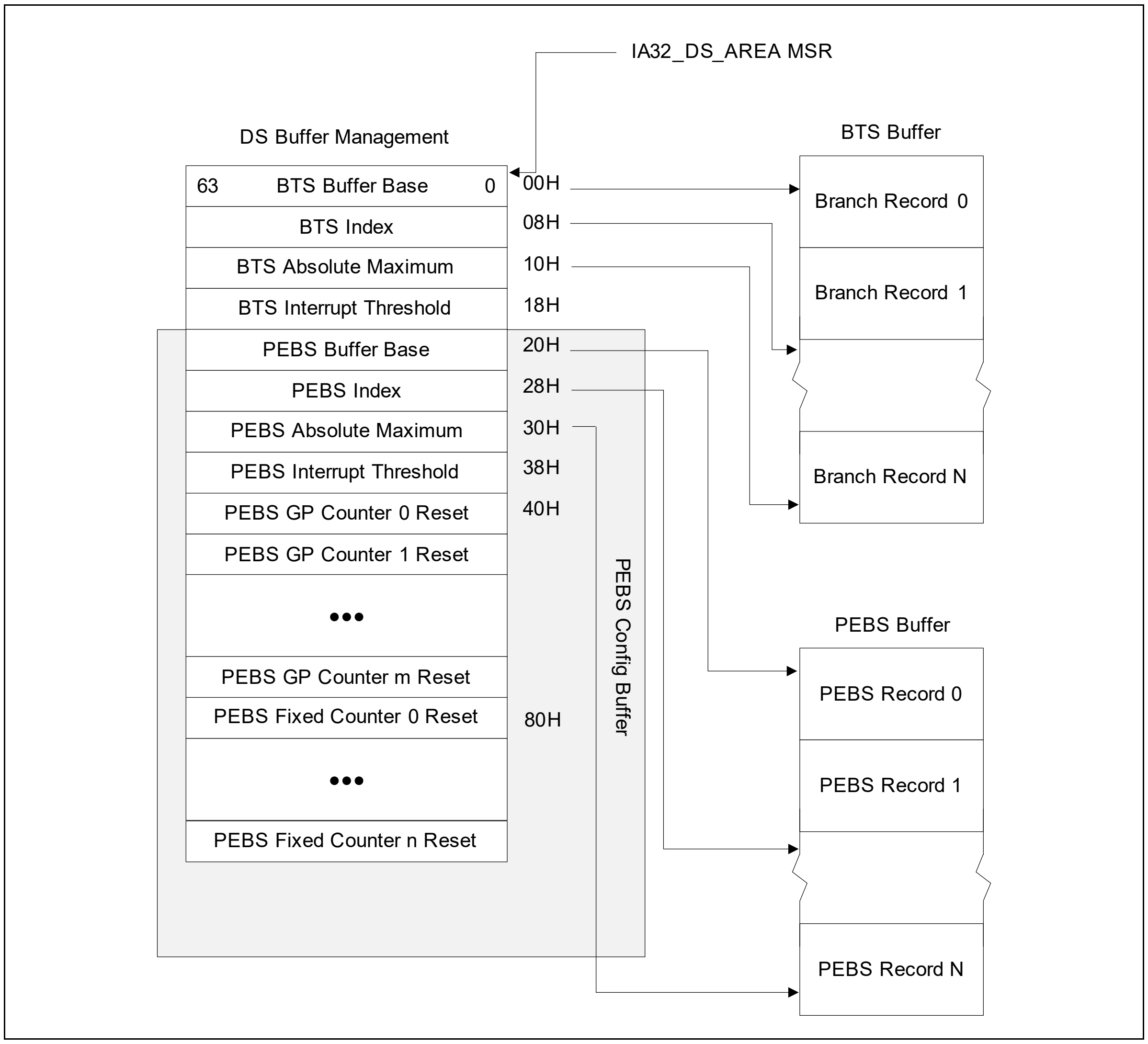


Figure 21-69. PEBS Programming Environment

Extended PEBS support debuts on Intel Atom® processors based on the Goldmont Plus microarchitecture and future Intel® Core™ processors based on the Ice Lake microarchitecture.

21.9.2 Adaptive PEBS

The PEBS facility has been enhanced to collect the following CPU state in addition to GPRs, EventingIP, TSC, and memory access related information collected by legacy PEBS:

- XMM registers
- LBR records (TO/FROM/INFO)
- Counters Snapshotting

The PEBS record is restructured where fields are grouped into Basic group, Memory group, GPR group, XMM group, LBR group, and Counters group. A new register MSR_PEBS_DATA_CFG provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.

By default, the PEBS record will only contain the Basic group. Optionally, each counter can be configured to generate a PEBS records with the groups specified in MSR_PEBS_DATA_CFG.

Details and examples for the Adaptive PEBS capability follow below.

21.9.2.1 Adaptive_Record Counter Control

IA32_PERFEVTSELx.Adaptive_Record[34]: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_PMCx is set for the corresponding GP counter, an overflow of PMCx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.

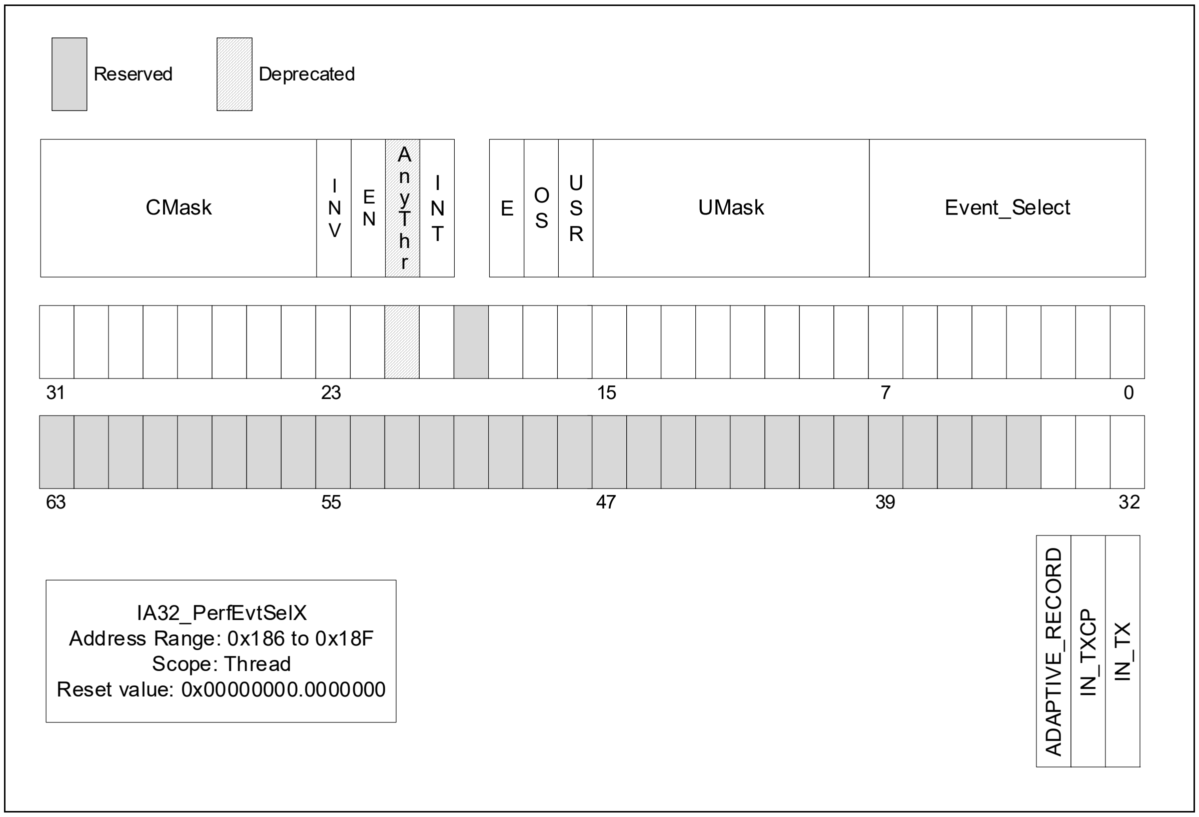


Figure 21-70. Layout of IA32_PerfEvtSelX MSR Supporting Adaptive PEBS

IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_FIXEDx is set for the corresponding Fixed counter, an overflow of FixedCtrx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.

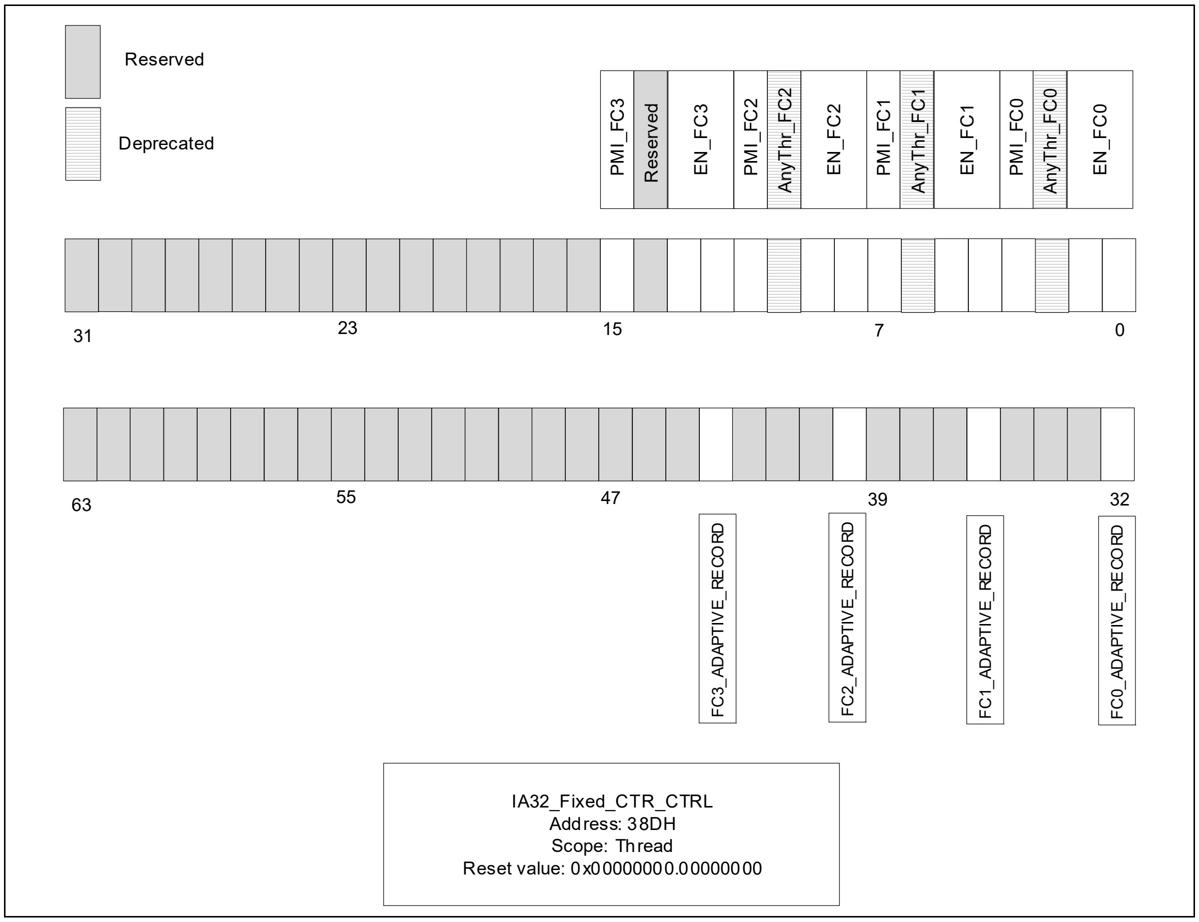


Figure 21-71. Layout of IA32_FIXED_CTR_CTRL MSR Supporting Adaptive PEBS

21.9.2.2 PEBS Record Format

The data fields in the PEBS record are aggregated into five groups which are described in the sub-sections below. Processors that support Adaptive PEBS implement a new MSR called MSR_PEBS_DATA_CFG which allows software to select the data groups to be captured. The data groups are not placed at fixed locations in the PEBS record, but are positioned immediately after one another, thus making the record format/size variable based on the groups selected.

21.9.2.2.1 Basic Info

The Basic group contains essential information for software to parse a record along with several critical fields. It is always collected.

Table 21-96. Basic Info Group

Field Name	Bit Width	Description
Record Format	[47:0]	This field indicates which data groups are included in the record. The field is zero if none of the counters that triggered the current PEBS record have their Adaptive_Record bit set. Otherwise it contains the value of MSR_PEBS_DATA_CFG.
	[63:48]	This field provides the size of the current record in bytes. Selected groups are packed back-to-back in the record without gaps or padding for unselected groups.

Table 21-96. Basic Info Group (Contd.)

Instruction Pointer	[63:0]	This field reports the Eventing Instruction Pointer (EventingIP) of the retired instruction that triggered the PEBS record generation. Note that this field is different than R/EIP which records the instruction pointer of the next instruction to be executed after record generation. The legacy R/EIP field has been removed.
Applicable Counters	[63:0]	The Applicable Counters field indicates which counters triggered the generation of the PEBS record, linking the record to specific events. This allows software to correlate the PEBS record entry properly with the instruction that caused the event, even when multiple counters are configured to generate PEBS records and multiple bits are set in the field.
TSC	[63:0]	This field provides the time stamp counter value when the PEBS record was generated.

21.9.2.2.2 Memory Access Info

This group contains the legacy PEBS memory-related fields; see Section 21.3.1.1.2.

Table 21-97. Memory Access Info Group

Field Name	Bit Width	Description
Memory Access Address	[63:0]	This field contains the linear address of the source of the load, or linear address of the destination (target) of the store. This value is written as a 64-bit address in canonical form.
Memory Auxiliary Info	[63:0]	When a MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains an encoded value indicating the memory hierarchy source which satisfied the load. These encodings are detailed in Table 21-5 and Table 21-14. If the PEBS assist was triggered for a store uop, this field will contain information indicating the status of the store, as detailed in Table 21-15.
Memory Access Latency[1]	[63:0]	When a MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains the latency to service the load in core clock cycles.
TSX Auxiliary Info	[31:0]	This field contains the number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
	[63:32]	This field contains the abort details. Refer to Section 21.3.6.5.1.

NOTES:

1. In certain conditions, high latencies in fields under "Memory Access Latency" may be observed even when the Data Src of the "Memory Auxiliary Info" field indicates a close source.

Beginning with 12th generation Intel Core processors, the memory access information group has been updated. New fields added are shaded gray in Table 21-98.

Table 21-98. Updated Memory Access Info Group

Field Name	Sub-field Name	Bits	Description
Access Address (offset 0H)	DLA	[63:0]	This field reports the data linear address (DLA) of the memory access in canonical form. A zero value indicates the processor could not retrieve the address of the particular access.
Access Info (offset 8H)	Data Src	[3:0]	An encoded value indicating the memory hierarchy source which satisfied the access. These encodings are detailed in Table 21-5. A zero value indicates the processor could not retrieve the data source of the particular access.
	STLB-miss	[4]	A value of 1 indicates the access has missed the Second-level TLB (STLB).
	Is-Lock	[5]	A value of 1 indicates the access was part of a locked (atomic) memory transaction.
	Data-Blk	[6]	A value of 1 indicates the load was blocked since its data could not be forwarded from a preceding store.
	Address-Blk	[7]	A value of 1 indicates the load was blocked due to potential address conflict with a preceding store.
Access Latency (offset 10H)	Instruction Latency	[15:0]	Measured instruction latency in core cycles. For loads, the latency starts by the dispatch of the load operation for execution and lasts until completion of the instruction it belongs to. This field includes the entire latency including time for data-dependency resolution or TLB lookups.
	Cache Latency	[47:32]	Measured cache access latency in core cycles. For loads, the latency starts by the actual cache access until the data is returned by the memory subsystem. For stores, the latency starts when the demand write accesses the L1 data-cache and lasts until the cacheline write is completed in the memory subsystem. This field does not include non-data-cache latency such as memory ordering checks or TLB lookups.
TSX (offset 18H)	Transaction Latency	[31:0]	This field contains the number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
	Abort Info	[63:32]	This field contains the abort details. Refer to Section 21.3.6.5.1.

To determine which fields are supported for certain performance monitoring events, consult the Memory Info attribute in the event lists at https://download.01.org/perfmon/.

NOTE

There may be additional block reasons, even if Data-Blk and Address-Blk are both clear, e.g., non-optimal instruction latency.

On P-core, the new Data-Blk and Address-Blk bits require the event LD_BLOCKS.STORE_FORWARD (r8203) to be configured in a programmable counter.

21.9.2.2.3 GPRs

This group is captured when the GPR bit is enabled in MSR_PEBS_DATA_CFG. GPRs are always 64 bits wide. If they are selected for non 64-bit mode, the upper 32-bit of the legacy RAX - RDI and all contents of R8-15 GPRs will be filled with 0s. In 64bit mode, the full 64 bit value of each register is written.

The order differs from legacy. The table below shows the order of the GPRs in Ice Lake microarchitecture.

Table 21-99. GPRs in Ice Lake Microarchitecture

Field Name	Bit Width
RFLAGS	[63:0]
RIP	[63:0]
RAX	[63:0]
RCX*	[63:0]
RDX*	[63:0]
RBX*	[63:0]
RSP*	[63:0]
RBP*	[63:0]
RSI*	[63:0]
RDI*	[63:0]
R8	[63:0]
...	...
R15	[63:0]

The machine state reported in the PEBS record is the committed machine state immediately after the instruction that triggers PEBS completes.

For instance, consider the following instruction sequence:

```
MOV eax, [eax]; triggers PEBS record generation
NOP
```

If the mov instruction triggers PEBS record generation, the EventingIP field in the PEBS record will report the address of the mov, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation. And the value of RIP will contain the linear address of the nop.

21.9.2.2.4 XMMs

This group is captured when the XMM bit is enabled in MSR_PEBS_DATA_CFG and SSE is enabled. If SSE is not enabled, the fields will contain zeroes. XMM8-XMM15 will also contain zeroes if not in 64-bit mode.

Table 21-100. XMMs

Field Name	Bit Width
XMM0	[127:0]
...	...
XMM15	[127:0]

21.9.2.2.5 LBRs

To capture LBR data in the PEBS record, the LBR bit in MSR_PEBS_DATA_CFG must be enabled. The number of LBR entries included in the record can be configured in the LBR_entries field of MSR_PEBS_DATA_CFG.

Table 21-101. LBRs

Field Name	Bit Width	Description
LBR[].FROM	[63:0]	Branch from address.
LBR[].TO	[63:0]	Branch to address.
LBR[].INFO	[63:0]	Other LBR information, like timing. This field is described in more detail in Section 19.12.1, "MSR_LBR_INFO_x MSR."

LBR entries are recorded into the record starting at LBR[TOS] and proceeding to LBR[TOS-1] and following. Note that LBR index is modulo the number of LBRs supporting on the processor.

21.9.2.3 MSR_PEBS_DATA_CFG

Bits in MSR_PEBS_DATA_CFG can be set to include data field blocks/groups into adaptive records. The Basic Info group is always included in the record. Additionally, the number of LBR entries included in the record is configurable.

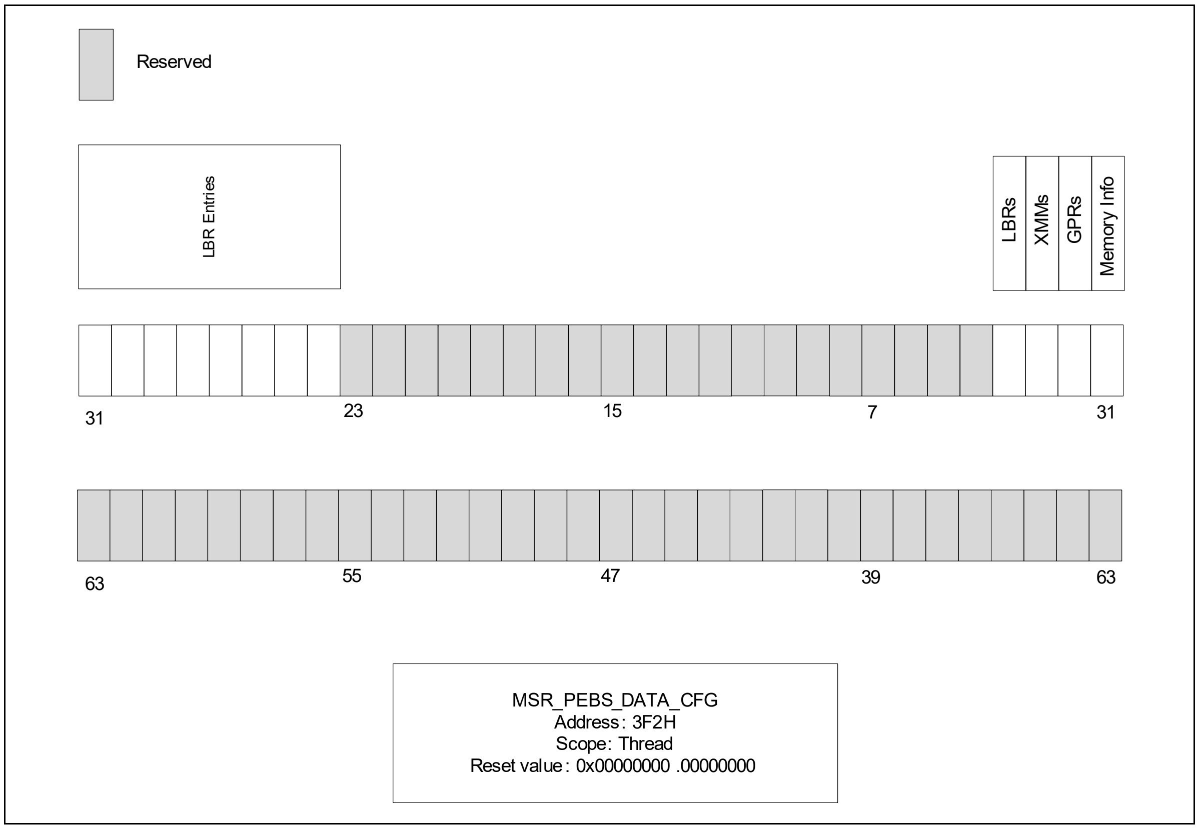


Figure 21-72. Legacy MSR_PEBS_DATA_CFG

Beginning with the Intel Series 2 Core Ultra processor, which counters are included in the Counters group is configurable. See Figure 21-73.

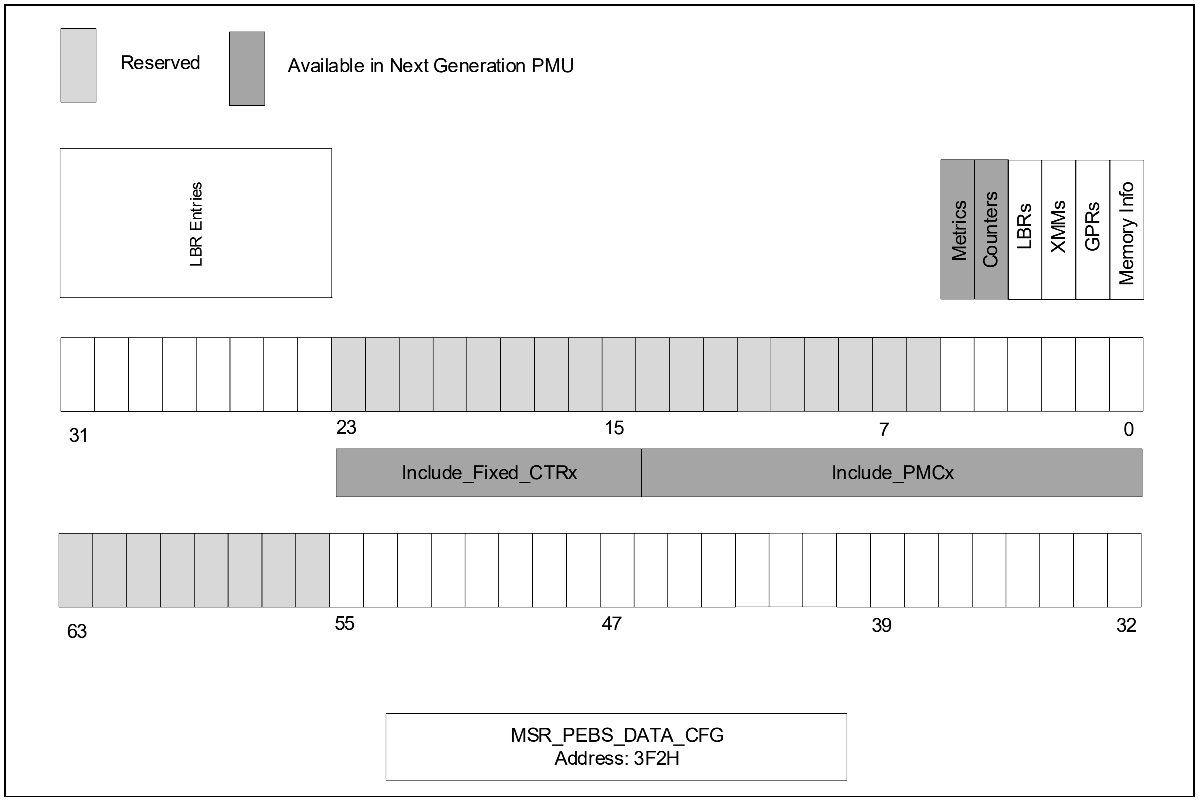


Figure 21-73. MSR_PEBS_DATA_CFG in PEBS_FMT=6

Table 21-102. MSR_PEBS_CFG Programming[1]

Bit Name	Bit Index	Access	Description	Availability
Memory Info	0	R/W	Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.	PEBS_FMT=4 and later
GPRs	1	R/W	Setting this bit will capture the contents of the General Purpose registers in the PEBS record.	PEBS_FMT=4 and later
XMMs	2	R/W	Setting this bit will capture the contents of the XMM registers in the PEBS record.	PEBS_FMT=4 and later
LBRs	3	R/W	Setting this bit will capture LBR TO, FROM, and INFO in the PEBS record.	PEBS_FMT=4 and later
Counters	4	R/W	Setting this bit will allow recording of the IA32_PMCx MSRs and the IA32_FIXED_CTRx counters. The Include_PMCx and Include_Fixed_CTRx bits are also set.	PEBS_FMT=6[2]
Metrics	5	R/W	Setting this bit will allow recording and clearing of the MSR_PERF_METRICS register (when the Include_Fixed_CTR3 bit is also set).	PEBS_FMT=6[2] && PERF_METRICS_AVAILABLE =1
Reserved[3]	23:6	NA	Reserved.	

Table 21-102. MSR_PEBS_CFG Programming[1] (Contd.)

LBR Entries	31:24	R/W	Set the field to the desired number of entries minus 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, it is recommended to select an even number of LBR entries (programmed into LBR_entries as an odd number).	PEBS_FMT=4 and later
Include_PMCx	47:32	R/W	A bit mask of the general-purpose counters that are allowed to be captured into the PEBS record. Note that only bits that match reporting of CPUID.(EAX=23H, ECX=01H):EAX are writable.	PEBS_FMT=6[2]
Include_FIXED_CTRx	55:48	R/W	A bit mask of the fixed-function counters that are allowed to be captured into the PEBS record. Note that only bits that match reporting of CPUID.(EAX=23H, ECX=01H):EBX are writable.	PEBS_FMT=6[2]
Reserved	63:56	NA	Reserved.	

NOTES:

1. A write to the MSR will be ignored when IA32_MISC_ENABLE.PERFMON_AVAILABLE is zero (default).
2. These fields are available starting with the IA32_PERF_CAPABILITIES.PEBS_FMT of 6 in addition to a subset of processors with a CPUID signature value of DisplayFamily_DisplayModel 06_C5H or 06_C6H (though they report IA32_PERF_CAPABILITIES.PEBS_FMT as 5).
3. Writing to the reserved bits will cause a GP fault.

21.9.2.3.6 Counters and Metrics Group

To capture the counters group, either the COUNTERS bit or the METRICS bit must be enabled in MSR_PEBS_DATA_CFG. The group allows recording of the IA32_PMCx MSRs, IA32_FIXED_CTRx MSRs, and the Performance Metrics.

The counters group first captures a 128-bit header with the bit vector of the counters that are captured later. The format of the counters header and the payload is shown in Table 21-103.

The group is available starting with IA32_PERF_CAPABILITIES.PEBS_FMT of 6. Additionally, the group is available in a subset of processors with a CPUID signature value of DisplayFamily_DisplayModel 06_C5H or 06_C6H (though they report IA32_PERF_CAPABILITIES.PEBS_FMT as 5).

Table 21-103. Counters Group

Field Name	Sub-Field Name	Bit Width	Description
Counters Group Header	PMC BitVector	[31:0]	Bit vector of IA32_PMCx MSRs. IA32_PMCx is recorded if bit x is set.
	FIXED_CTR BitVector	[31:0]	Bit vector of IA32_FIXED_CTRx MSRs. IA32_FIXED_CTRx is recorded if bit x is set.
	Metrics BitVector	[31:0]	Bit vector of the performance metrics counters.
	Reserved	[31:0]	Reserved.

Table 21-103. Counters Group (Contd.)

Field Name	Sub-Field Name	Bit Width	Description
Counters/Metrics Values	PMCx	[63:0]	PMCx will be captured if PMC BitVector x is set.
	...		
	FIXED CTRx	[63:0]	FIXED_CTRx will be captured if FIXED_CTRx BitVector x is set.
	...		
	Metrics Base	[63:0]	The performance metrics base, mapped to IA32_FIXED_CTR3, if Metrics BitVector bit 0 is set.
	Metrics Data	[63:0]	MSR_PERF_METRICS, if Metrics BitVector bit 1 is set.

IA32_PMCx will be captured if both Counters and MSR_PEBS_DATA_CFG bit 32 + x are set. In this case, the PMC BitVector field bit x will be set too.

IA32_FIXED_CTRx will be captured if both Counters and MSR_PEBS_DATA_CFG bit 48 + x are set. In this case, the FIXED_CTR BitVector field bit x will be set too.

The performance metrics will be recorded if both Metrics and MSR_PEBS_DATA_CFG bit 51 (the bit used for IA32_FIXED_CTR3) are set. The Metrics record will have two 64-bit fields, MSR_PERF_METRICS and the PERF_METRICS_BASE that is derived from IA32_FIXED_CTR3. In this case, the Metrics BitVector will be 3. Note that MSR_PERF_METRICS and the IA32_FIXED_CTR3 MSR will be cleared after they are recorded.

Size of the group can be calculated in bytes by: 16 + popcount(BitVectors[127:0]) * 8.

21.9.2.4 PEBS Record Examples

The following example shows the layout of the PEBS record when all data groups are selected (all valid bits in MSR_PEBS_DATA_CFG are set) and maximum number of LBRs are selected. There are no gaps in the PEBS record when a subset of the groups are selected, thus keeping the layout compact. Implementations that do not support some features will have to pad zeroes in the corresponding fields.

Table 21-104. PEBS Record Example 1

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	
0x20	Memory Info	Memory Access Address	DLA
0x28		Memory Auxiliary Info	DATA_SRC
0x30		Memory Access Latency	Load Latency
0x38		TSX Auxiliary Info	HLE Information

Table 21-104. PEBS Record Example 1 (Contd.)

0x40	GPRs	RFLAGS	
0x48		RIP	
0x50		RAX	
...		...	
0x88		RDI	
0x90		R8	
...		...	
0xC8		R15	
0xD0	XMMs	XMM0	New
...		...	
0x1C0		XMM15	
0x1D0	LBRs	LBR[TOS].FROM	New
0x1D8		LBR[TOS].TO	
0x1E0		LBR[TOS].INFO	
...		...	
0x4B8		LBR[TOS +1].FROM	
0x4C0		LBR[TOS +1].TO	
0x4C8		LBR[TOS +1].INFO	

The following example shows the layout of the PEBS record when Basic, GPR, and LBR group with 3 LBR entries are selected.

Table 21-105. PEBS Record Example 2

Offset	Group Name	Field Name	Legacy Name (If Different)
0x0	Basic Info	Record Format	New
		Record Size	New
0x8		Instruction Pointer	EventingRIP
0x10		Applicable Counters	
0x18		TSC	
0x20	GPRs	RFLAGS	
0x28		RIP	
0x30		RAX	
...		...	
0x68		RDI	
0x70		R8	
...		...	
0xA8		R15	
0xB0	LBRs	LBR[TOS].FROM	New
0xB8		LBR[TOS].TO	
0xC0		LBR[TOS].INFO	
...		...	
0xE0		LBR[TOS +1].FROM	
0xE8		LBR[TOS +1].TO	
0xF0		LBR[TOS +1].INFO	

21.9.3 Precise Distribution of Instructions Retired (PDIR) Facility

Precise Distribution of Instructions Retired Facility is available via PEBS on some microarchitectures. Refer to Section 21.3.4.4.4. Counters that support PDIR also vary. See the processor specific sections for availability.

21.9.4 Reduced Skid PEBS

For precise events, upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. The Reduced Skid mechanism mitigates the “skid” problem by providing an early indication of when the counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus greatly reducing skid.

This mechanism is a superset of the PDIR mechanism available in the Sandy Bridge microarchitecture. See Section 21.3.4.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including, INST_RETIRED, except for UOPS_RETIRED. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

With Reduced Skid PEBS, the skid is precisely one event occurrence. Hence if counting INST_RETIRED, PEBS will indicate the instruction that follows that which caused the counter to overflow.

For the Reduced Skid mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g., via IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g., what event is specified in IA32_PERFEVTSELx or whether PEBS is enabled for that counter via IA32_PEBS_ENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

21.9.5 EPT-Friendly PEBS

The 3rd generation Intel Xeon Scalable Family of processors based on Ice Lake microarchitecture (and later processors) and the 12th generation Intel Core processor (and later processors) support VMX guest use of PEBS when the DS Area (including the PEBS Buffer and DS Management Area) is allocated from a paged pool of EPT pages. In such a configuration PEBS DS Area accesses may result in VM exits (e.g., EPT violations due to "lazy" EPT page-table entry propagation), and in such cases the PEBS record will not be lost but instead will "skid" to after the subsequent VM Entry back to the guest. For precise events the guest will observe that the record skid by one event occurrence, while for non-precise events the record will skid by one instruction.

21.9.6 PDist: Precise Distribution

PDist eliminates any skid or shadowing effects from PEBS. With PDist, the PEBS record will be generated precisely upon completion of the instruction or operation that causes the counter to overflow (there is no "wait for next occurrence" by default).

PDist is supported by selected counters, and is only supported when those counters are programmed to count select precise events[1]. The legacy PEBS behavior applies to counters that do not support PDist, unless specified otherwise. PDist requires that the INV, ANY, E, EQ, and CMASK fields are cleared. Which counters support PDist, and which events are supported for PDist, is model-specific. Further, the counter reload value must not be less than 256 for PDist to operate.

For the PDist mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g., via IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g., what event is specified in IA32_PERFEVTSELx or IA32_FIXED_CTR_CTRL or whether PEBS is enabled for that counter via IA32_PEBS_ENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx or IA32_FIXED_CTRx).

21.9.7 Load Latency Facility

The load latency facility provides software a means to characterize the latencies of memory load operations to different levels of cache/memory hierarchy. This facility requires a processor supporting the enhanced PEBS record format in the PEBS buffer.

Beginning with 12th generation Intel Core processors, the load latency facility supports all fields in Table 21-98, "Updated Memory Access Info Group," in addition to the Memory Access Address field:

- The **Instruction Latency** field measures the load latency from the load's first dispatch until final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches and data dependencies).
- The **Cache Latency** field measures the subset of cache access latency in core cycles. It starts from the actual cache access until the data is returned by the memory subsystem The latency is reported for retired demand load operations in core cycles (it does not account for memory ordering blocks).
- The **Data Source** field is an encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 21-106. In the descriptions, local memory refers to system memory physically

1. To determine whether an event is precise or supports PDist, consult the relevant attribute in the event lists at https://download.01.org/perfmon/.

attached to a processor package, and remote memory refers to system memory or cache physically attached to another processor package (in a server product).

- Through the **Access Info** field, load latency features binary indications on certain blocks that the load operation may have encountered. Refer to STLB-miss, Is-Lock, Data-Blk and Address-Blk fields in Table 21-98.

NOTE

For loads triggered by software prefetch instructions, the cache related fields including Data Source and Cache Latency, report values as if the load was an L1 cache hit (the prefetch completes without waiting for data return, for performance reasons).

Table 21-106. Data Source Encoding for Memory Accesses (Ice Lake and Later Microarchitectures)

Encoding [3:0]	Description
00H	Unknown Data Source (the processor could not retrieve the origin of this request).
01H	L1 HIT. This request was satisfied by the L1 data cache. (Minimal latency core cache hit.)
02H	FB HIT. This request was merged into an outstanding cache miss to same cache-line address.
03H	L2 HIT. This request was satisfied by the L2 cache.
04H	L3 HIT. This request was satisfied by the L3 cache with no coherency actions performed (snooping).
05H	XCORE MISS. This request was satisfied by the L3 cache but involved a coherency check in some sibling core(s).
06H	XCORE HIT. This request was satisfied by the L3 cache but involved a coherency check that hit a non-modified copy in a sibling core.
07H	XCORE FWD. This request was satisfied by a sibling core where either a modified (cross-core HITM) or a non-modified (cross-core FWD) cache-line copy was found.
08H	Local Far Memory. This request has missed the L3 cache and was serviced by local far memory.
09H	Remote Far Memory. This request has missed the L3 cache and was serviced by remote far memory.
0AH	Local Near Memory. This request has missed the L3 cache and was serviced by local near memory.
0BH	Remote Near Memory. This request has missed the L3 cache and was serviced by remote near memory.
0CH	Remote FWD. This request has missed the L3 cache and a non-modified cache-line copy was forwarded from a remote cache.
0DH	Remote HITM. This request has missed the L3 cache and a modified cache-line was forwarded from a remote cache.
0EH	I/O. Request of input/output operation.
0FH	UC. The request was to uncacheable memory.

Table 21-107. Data Source Encoding for Memory Accesses (Lion Cove and Next Generation Microarchitectures)

Encoding [4:0]	Description
00H	Unknown Data Source (the processor could not retrieve the origin of this request).
01H or 02H	L1 HIT. This request was satisfied by the L1 data cache. (Minimal latency core cache hit.)
03H	FB merge. L1 mishandling buffer.
05H	L2 HIT. This request was satisfied by the L2 cache.
06H	XQ merge. L2 mishandling buffer.
08H	L3 HIT. This request was satisfied by the L3 cache.
0CH	L3 Hit, x-core forward.
0DH	L3 Hit, x-core modified.
0FH	L3 Miss, x-core modified.

Table 21-107. Data Source Encoding for Memory Accesses (Lion Cove and Next Generation Microarchitectures)

Encoding [4:0]	Description
10H	L3 Miss, MSC Hit (memory-side cache).
11H	L3 Miss, memory.

To use this feature, software must complete the following steps:

- Complete the PEBS configuration steps.
- Set the Memory Info bit in the PEBS_DATA_CFG MSR.
- One of the relevant IA32_PERFEVTSELx MSRs is programmed to specify the event unit MEM_TRANS_RETIRED.LOAD_LATENCY (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter, IA32_PMCx, will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in an MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with instruction latency greater than this value are eligible for counting and PEBS data reporting. The minimum value that may be programmed in this register is 1.
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register.

Refer to Section 21.3.4.4.2 for further implementation details of Load Latency.

21.9.8 Store Latency Facility

Store latency support is available on the 12th generation Intel Core processor. Store latency is a PEBS extension that provides a means to profile store memory accesses in the system. It complements the load latency facility.

Store latency leverages the PEBS facility where it can provide additional information about sampled stores. The additional information includes the data address, memory auxiliary information, and the cache latency of the store access. Normal stores (those preceded with a read-for-ownership) as well as streaming stores are supported by the store latency facility.

Memory store operations typically do not limit performance since they update the memory with no operation that directly depends on them. Thus, data out of this facility should be carefully used once stores are suspected as a performance limiter; for example, once the TMA node of Backend_Bound.Memory_Bound.Store_Bound is flagged[1].

To enable the store latency facility, software must complete the following steps:

- Complete the PEBS configuration steps.
- Set the Memory Info bit in the PEBS_DATA_CFG MSR.
- Program the MEM_TRANS_RETIRED.STORE_SAMPLE event on general-purpose performance-monitoring counter 0 (IA32_PERFEVTSEL0[15:0] = 2CDH).
- Setup the PEBS buffer to hold at least two records, setting both 'PEBS Absolute Maximum' and 'PEBS Interrupt Threshold', should any other counter be used by PEBS (that is whenever IA32_PEBS_ENABLE[x] $\neq$ 0 for x $\neq$ 0).
- Set IA32_PEBS_ENABLE[0].

The store latency information is written into a PEBS record as shown in Table 21-49.

The store latency relies on the PEBS facility, so the PEBS configuration must be completed first. Unlike load latency, there is no option to filter on a subset of stores that exceed a certain threshold.

1. For more details about the method, refer to Section B.1, "Top-Down Analysis Method" of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

21.9.9 Timed Processor Event Based Sampling

Timed Processor Event Based Sampling (Timed PEBS) enables recording of time in every PEBS record. It extends all PEBS records with timing information in a new “Retire Latency” field that is placed in the Basic Info group of the PEBS record as shown in Table 21-108.

Table 21-108. PEBS Basic Info Group

Offset	Field Name	Bits
0x0	Record Format	[31:0]
	Retire Latency	[47:32]
	Record Size	[63:48]
0x08	Instruction Pointer	[63:0]
0x10	Applicable Counters	[63:0]
0x18	TSC	[63:0]

The Retire Latency field reports the number of Unhalted Core Cycles between the retirement of the current instruction (as indicated by the Instruction Pointer field of the PEBS record) and the retirement of the prior instruction. All ones are reported when the number exceeds 16 bits.

Processors that support this enhancement set a new bit: IA32_PERF_CAPABILITIES.PEBS_TIMING_INFO[bit 17].

NOTE

Timed PEBS is not supported when PEBS is programmed on fixed-function counter 0. The Retire Latency field of such record is undefined.

21.9.10 Counters Snapshotting

Counters Snapshotting extends Adaptive PEBS with the PEBS Counters and Metrics group. This extension enables software to capture general-purpose counters, fixed-function counters, and performance metrics in the PEBS record. For additional details, see Section 21.9.2.3.6, “Counters and Metrics Group.”

21.10 AUTO COUNTER RELOAD

Auto Counter Reload (ACR) provides a means for software to specify that, for each supported counter, the hardware should automatically reload the counter to a specified initial value upon overflow of chosen counters. This mechanism enables software to sample based on the relative rate of two (or more) events, such that a sample (PMI or PEBS) is taken only if the rate of one event exceeds some threshold relative to the rate of another event. Taking a PMI or PEBS only when the relative rate of performance-monitoring events crosses a threshold can have significantly less performance overhead than other techniques (e.g., taking a PMI every 1000 instructions in order to check the number of mispredicts since the last PMI).

21.10.1 Discovery and Interface

CPUID.(EAX=23H, ECX=02H):EAX indicates general-purpose counters [n:0] that can be reloaded.
CPUID.(EAX=23H, ECX=02H):EBX indicates fixed-function counters [m:0] that can be reloaded.
CPUID.(EAX=23H, ECX=02H):ECX indicates general-purpose counters [n:0] that can cause a reload of reloadable counters. CPUID.(EAX=23H, ECX=02H):EDX indicates fixed-function counters [m:0] that can cause a reload of reloadable counters. If a counter can be reloaded, its associated reload configuration MSR (*_CFG_B) and its reload value MSR (*_CFG_C) are supported.

See Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for details about the following MSRs: IA32_PMC_GPn_CFG_B, IA32_PMC_GPn_CFG_C, IA32_PMC_FXm_CFG_B, and IA32_PMC_FXm_CFG_C.

21.10.2 Configuration and Behavior

For a given counter IA32_PMC_GPn_CTR, bit fields in the IA32_PMC_GPn_CFG_B MSR indicate which counter(s) can cause a reload of that counter:

- If the general-purpose counter 'n' is configured to do a reload when general-purpose counter 'x' overflows (IA32_PMC_GPn_CFG_B.PMC[x] = 1), then that general-purpose counter 'n' will be written with its reload value (in IA32_PMC_GPn_CFG_C[31:0]) when counter 'x' (IA32_PMC_GPx_CTR) overflows.
- If general-purpose counter 'n' is configured to do a reload when fixed-function counter 'x' overflows (IA32_PMC_GPn_CFG_B.FIXED_CTR[x] = 1), then that general-purpose counter 'n' will be written with its reload value (in IA32_PMC_GPn_CFG_C[31:0]) when fixed-function counter 'x' (IA32_PMC_FXx_CTR) overflows.

ACR will not reload IA32_PMC_GPn_CTR if counters are frozen (IA32_PERF_GLOBAL_STATUS.COUNTERS_FROZEN = 1) or if IA32_PMC_GPn_CTR has already overflowed (IA32_PERF_GLOBAL_STATUS.PMCn_OVF = 1). If a PMI or PEBS is taken due to a counter overflow, the PMI ISR or PEBS record can record the unmodified counter value before reloading the counter. In race conditions, where IA32_PMC_GPn_CTR overflows in the same cycle as a counter configured to reload the IA32_PMC_GPn_CTR on overflow, IA32_PMC_GPn_CTR will not be reloaded, and IA32_PERF_GLOBAL_STATUS.PMCn_OVF will be set.

For counters that reload themselves (i.e., IA32_PMC_GPn_CFG_B.PMCn = 1), the overflow bit (IA32_PERF_GLOBAL_STATUS.PMCn_OVF) will never be set. Instead, upon overflow, the counter will be immediately reloaded; thus, it is never in an overflowed state. There is an exception associated with PEBS; see Section 21.10.2.2.

The behavior is similar for reloading of fixed-function counters. For IA32_PMC_FXm_CTR, the reload value is stored in IA32_PMC_FXm_CFG_C[31:0], and which counters cause reload of IA32_PMC_FXm_CTR is configured in IA32_PMC_FXm_CFG_B.

21.10.2.1 Reload Precision

ACR reload is not guaranteed to be precise; in some cases, a small number of events may be lost during the time between counter overflow and counter reload. However, when the reload happens, hardware will reload all configured counters simultaneously.

21.10.2.2 PEBS Interaction

If a counter is configured to reload other counters with ACR and to take PEBS on overflow, the counter reload actions will be taken only after the PEBS record has been written. This ensures that any counter values captured in the PEBS record reflect the value before the reload occurs. Because the reload actions are taken after the PEBS records are written, reloaded counter value will not account for the events that occurred during the process of writing the PEBS record.

For a counter configured to reload itself and to take PEBS on overflow, the overflow bit associated with the counter (in IA32_PERF_GLOBAL_STATUS) will be set from the time the counter overflows to the time the PEBS record is written. This is required to ensure the PEBS record is not lost due to a VM exit taken during record generation. Once the record is written, the overflow bit will be cleared, and the counter reloaded.

21.10.2.3 Precise Distribution (PDIST) Interaction

Precise distribution of PEBS events (PDIR) is not supported when such a counter is reloaded by ACR. For details on PDIST, see Section 21.9.6.

12. Updates to Chapter 26, Volume 3C

Change bars and violet text show changes to Chapter 26 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C:* System Programming Guide, Part 3.

Changes to this chapter:

- Added information for the execution of WRMSRLIST to Section 26.9.4, "Information for VM Exits Due to Instruction Execution."

CHAPTER 26
VIRTUAL MACHINE CONTROL STRUCTURES

26.1 OVERVIEW

A logical processor uses **virtual-machine control data structures** (**VMCSs**) while it is in VMX operation. These manage transitions into and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. This structure is manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE.

A VMM can use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM can use a different VMCS for each virtual processor.

A logical processor associates a region in memory with each VMCS. This region is called the **VMCS region**.[1] Software references a specific VMCS using the 64-bit physical address of the region (a **VMCS pointer**). VMCS pointers must be aligned on a 4-KByte boundary (bits 11:0 must be zero). These pointers must not set bits beyond the processor's physical-address width.[2,3]

A logical processor may maintain a number of VMCSs that are **active**. The processor may optimize VMX operation by maintaining the state of an active VMCS in memory, on the processor, or both. At any given time, at most one of the active VMCSs is the **current** VMCS. (This document frequently uses the term "the VMCS" to refer to the current VMCS.) The VMLAUNCH, VMREAD, VMRESUME, and VMWRITE instructions operate only on the current VMCS.

The following items describe how a logical processor determines which VMCSs are active and which is current:

- The memory operand of the VMPTRLD instruction is the address of a VMCS. After execution of the instruction, that VMCS is both active and current on the logical processor. Any other VMCS that had been active remains so, but no other VMCS is current.
- The VMCS link pointer field in the current VMCS (see Section 26.4.2) is itself the address of a VMCS. If VM entry is performed successfully with the 1-setting of the "VMCS shadowing" VM-execution control, the VMCS referenced by the VMCS link pointer field becomes active on the logical processor. The identity of the current VMCS does not change.
- The memory operand of the VMCLEAR instruction is also the address of a VMCS. After execution of the instruction, that VMCS is neither active nor current on the logical processor. If the VMCS had been current on the logical processor, the logical processor no longer has a current VMCS.

The VMPTRST instruction stores the address of the logical processor's current VMCS into a specified memory location (it stores the value FFFFFFFF_FFFFFFFFH if there is no current VMCS).

The **launch state** of a VMCS determines which VM-entry instruction should be used with that VMCS: the VMLAUNCH instruction requires a VMCS whose launch state is "clear"; the VMRESUME instruction requires a VMCS whose launch state is "launched". A logical processor maintains a VMCS's launch state in the corresponding VMCS region. The following items describe how a logical processor manages the launch state of a VMCS:

- If the launch state of the current VMCS is "clear", successful execution of the VMLAUNCH instruction changes the launch state to "launched".
- The memory operand of the VMCLEAR instruction is the address of a VMCS. After execution of the instruction, the launch state of that VMCS is "clear".
- There are no other ways to modify the launch state of a VMCS (it cannot be modified using VMWRITE) and there is no direct way to discover it (it cannot be read using VMREAD).

1. The amount of memory required for a VMCS region is at most 4 KBytes. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix A.1).
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. If IA32_VMX_BASIC[48] is read as 1, these pointers must not set any bits in the range 63:32; see Appendix A.1.

Figure 26-1 illustrates the different states of a VMCS. It uses "X" to refer to the VMCS and "Y" to refer to any other VMCS. Thus: "VMPTRLD X" always makes X current and active; "VMPTRLD Y" always makes X not current (because it makes Y current); VMLAUNCH makes the launch state of X "launched" if X was current and its launch state was "clear"; and VMCLEAR X always makes X inactive and not current and makes its launch state "clear".

The figure does not illustrate operations that do not modify the VMCS state relative to these parameters (e.g., execution of VMPTRLD X when X is already current). Note that VMCLEAR X makes X "inactive, not current, and clear," even if X's current state is not defined (e.g., even if X has not yet been initialized). See Section 26.11.3.

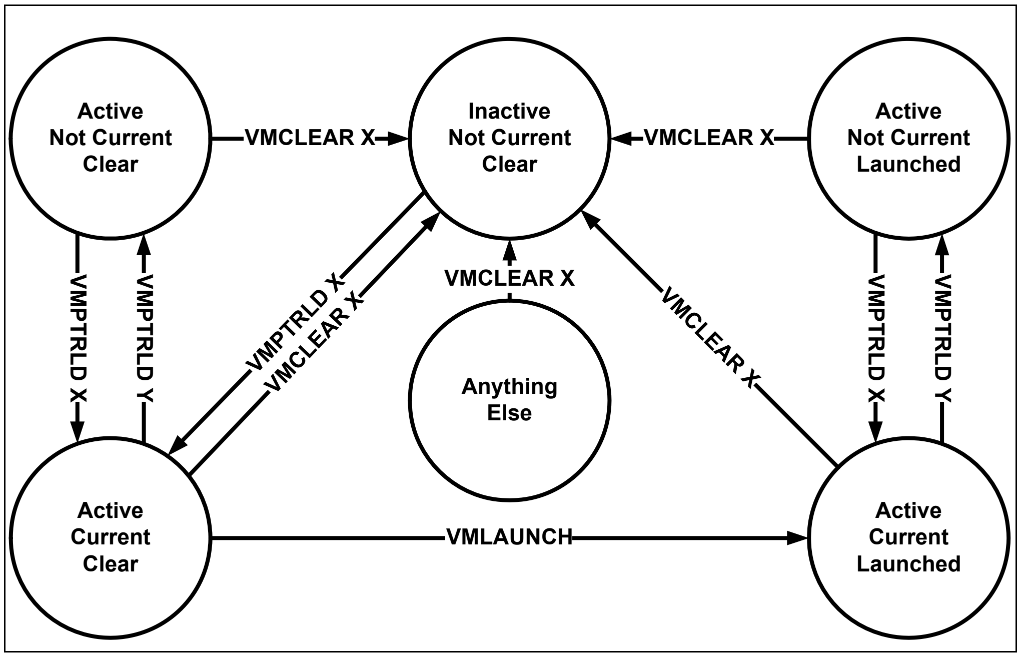


Figure 26-1. States of VMCS X

Because a shadow VMCS (see Section 26.10) cannot be used for VM entry, the launch state of a shadow VMCS is not meaningful. Figure 26-1 does not illustrate all the ways in which a shadow VMCS may be made active.

26.2 FORMAT OF THE VMCS REGION

A VMCS region comprises up to 4-KBytes.[1] The format of a VMCS region is given in Table 26-1.

Table 26-1. Format of the VMCS Region

Byte Offset	Contents
0	Bits 30:0: VMCS revision identifier Bit 31: shadow-VMCS indicator (see Section 26.10)
4	VMX-abort indicator
8	VMCS data (implementation-specific format)

1. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix A.1).

The first 4 bytes of the VMCS region contain the **VMCS revision identifier** at bits 30:0.[1] Processors that maintain VMCS data in different formats (see below) use different VMCS revision identifiers. These identifiers enable software to avoid using a VMCS region formatted for one processor on a processor that uses a different format.[2] Bit 31 of this 4-byte region indicates whether the VMCS is a shadow VMCS (see Section 26.10).

Software should write the VMCS revision identifier to the VMCS region before using that region for a VMCS. The VMCS revision identifier is never written by the processor; VMPTRLD fails if its operand references a VMCS region whose VMCS revision identifier differs from that used by the processor. (VMPTRLD also fails if the shadow-VMCS indicator is 1 and the processor does not support the 1-setting of the "VMCS shadowing" VM-execution control; see Section 26.6.2) Software can discover the VMCS revision identifier that a processor uses by reading the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

Software should clear or set the shadow-VMCS indicator depending on whether the VMCS is to be an ordinary VMCS or a shadow VMCS (see Section 26.10). VMPTRLD fails if the shadow-VMCS indicator is set and the processor does not support the 1-setting of the "VMCS shadowing" VM-execution control. Software can discover support for this setting by reading the VMX capability MSR IA32_VMX_PROCBASED_CTLS2 (see Appendix A.3.3).

The next 4 bytes of the VMCS region are used for the **VMX-abort indicator**. The contents of these bits do not control processor operation in any way. A logical processor writes a non-zero value into these bits if a VMX abort occurs (see Section 29.7). Software may also write into this field.

The remainder of the VMCS region is used for **VMCS data** (those parts of the VMCS that control VMX non-root operation and the VMX transitions). The format of these data is implementation-specific. VMCS data are discussed in Section 26.3 through Section 26.9. To ensure proper behavior in VMX operation, software should maintain the VMCS region and related structures (enumerated in Section 26.11.4) in writeback cacheable memory. Future implementations may allow or require a different memory type[3]. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

26.3 ORGANIZATION OF VMCS DATA

The VMCS data are organized into six logical groups:

- **Guest-state area.** Processor state is saved into the guest-state area on VM exits and loaded from there on VM entries.
- **Host-state area.** Processor state is loaded from the host-state area on VM exits.
- **VM-execution control fields.** These fields control processor behavior in VMX non-root operation. They determine in part the causes of VM exits.
- **VM-exit control fields.** These fields control VM exits.
- **VM-entry control fields.** These fields control VM entries.
- **VM-exit information fields.** These fields receive information on VM exits and describe the cause and the nature of VM exits. On some processors, these fields are read-only.[4]

The VM-execution control fields, the VM-exit control fields, and the VM-entry control fields are sometimes referred to collectively as VMX controls.

1. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field. For all processors produced prior to this change, bit 31 of the VMCS revision identifier was 0.
2. Logical processors that use the same VMCS revision identifier use the same size for VMCS regions.
3. Alternatively, software may map any of these regions or structures with the UC memory type. Doing so is strongly discouraged unless necessary as it will cause the performance of transitions using those structures to suffer significantly. In addition, the processor will continue to use the memory type reported in the VMX capability MSR IA32_VMX_BASIC with exceptions noted in Appendix A.1.
4. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

26.4 GUEST-STATE AREA

This section describes fields contained in the guest-state area of the VMCS. VM entries load processor state from these fields and VM exits store processor state into these fields. See Section 28.3.2 and Section 29.3 for details.

26.4.1 Guest Register State

The following fields in the guest-state area correspond to processor registers:

- Control registers CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- Debug register DR7 (64 bits; 32 bits on processors that do not support Intel 64 architecture).
- RSP, RIP, and RFLAGS (64 bits each; 32 bits on processors that do not support Intel 64 architecture).[1]
- The following fields for each of the registers CS, SS, DS, ES, FS, GS, LDTR, and TR:
  - Selector (16 bits).
  - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture). The base-address fields for CS, SS, DS, and ES have only 32 architecturally-defined bits; nevertheless, the corresponding VMCS fields have 64 bits on processors that support Intel 64 architecture.
  - Segment limit (32 bits). The limit field is always a measure in bytes.
  - Access rights (32 bits). The format of this field is given in Table 26-2 and detailed as follows:
    - The low 16 bits correspond to bits 23:8 of the upper 32 bits of a 64-bit segment descriptor. While bits 19:16 of code-segment and data-segment descriptors correspond to the upper 4 bits of the segment limit, the corresponding bits (bits 11:8) are reserved in this VMCS field.
    - Bit 16 indicates an **unusable segment**. Attempts to use such a segment fault except in 64-bit mode. In general, a segment register is unusable if it has been loaded with a null selector.[2]
    - Bits 31:17 are reserved.

Table 26-2. Format of Access Rights

Bit Position(s)	Field
3:0	Segment type
4	S — Descriptor type (0 = system; 1 = code or data)
6:5	DPL — Descriptor privilege level
7	P — Segment present
11:8	Reserved
12	AVL — Available for use by system software

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

2. There are a few exceptions to this statement. For example, a segment with a non-null selector may be unusable following a task switch that fails after its commit point; see “Interrupt 10—Invalid TSS Exception (#TS)” in Section 7.14, “Exception and Interrupt Handling in 64-bit Mode,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. In contrast, the TR register is usable after processor reset despite having a null selector; see Table 12-1 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Table 26-2. Format of Access Rights (Contd.)

Bit Position(s)	Field
13	Reserved (except for CS) L — 64-bit mode active (for CS only)
14	D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
15	G — Granularity
16	Segment unusable (0 = usable; 1 = unusable)
31:17	Reserved

The base address, segment limit, and access rights compose the "hidden" part (or "descriptor cache") of each segment register. These data are included in the VMCS because it is possible for a segment register's descriptor cache to be inconsistent with the segment descriptor in memory (in the GDT or the LDT) referenced by the segment register's selector.

The value of the DPL field for SS is always equal to the logical processor's current privilege level (CPL).[1]

On some processors, executions of VMWRITE ignore attempts to write non-zero values to any of bits 11:8 or bits 31:17. On such processors, VMREAD always returns 0 for those bits, and VM entry treats those bits as if they were all 0 (see Section 28.3.1.2).

- The following fields for each of the registers GDTR and IDTR:
  - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture).
  - Limit (32 bits). The limit fields contain 32 bits even though these fields are specified as only 16 bits in the architecture.
- The following MSRs:
  - IA32_DEBUGCTL (64 bits)
  - IA32_SYSENTER_CS (32 bits)
  - IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture)
  - IA32_PERF_GLOBAL_CTRL (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-entry control.
  - IA32_PAT (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_PAT" VM-entry control or that of the "save IA32_PAT" VM-exit control.
  - IA32_EFER (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_EFER" VM-entry control or that of the "save IA32_EFER" VM-exit control.
  - IA32_BNDCFGS (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_BNDCFGS" VM-entry control or that of the "clear IA32_BNDCFGS" VM-exit control.
  - IA32_RTIT_CTL (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_RTIT_CTL" VM-entry control or that of the "clear IA32_RTIT_CTL" VM-exit control.
  - IA32_LBR_CTL (64 bits). This field is supported only on processors that support either the 1-setting of the "load guest IA32_LBR_CTL" VM-entry control or that of the "clear IA32_LBR_CTL" VM-exit control.
  - IA32_S_CET (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-entry control.
  - IA32_INTERRUPT_SSP_TABLE_ADDR (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-entry control.

1. In protected mode, CPL is also associated with the RPL field in the CS selector. However, the RPL fields are not meaningful in real-address mode or in virtual-8086 mode.

— IA32_PKRS (64 bits). This field is supported only on processors that support the 1-setting of the "load PKRS" VM-entry control.

- The shadow-stack pointer register SSP (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-entry control.
- The register SMBASE (32 bits). This register contains the base address of the logical processor's SMRAM image.

26.4.2 Guest Non-Register State

In addition to the register state described in Section 26.4.1, the guest-state area includes the following fields that characterize guest state but which do not correspond to processor registers:

- **Activity state** (32 bits). This field identifies the logical processor's activity state. When a logical processor is executing instructions normally, it is in the **active state**. Execution of certain instructions and the occurrence of certain events may cause a logical processor to transition to an **inactive state** in which it ceases to execute instructions.

  The following activity states are defined:[1]

  — 0: **Active**. The logical processor is executing instructions normally.

  — 1: **HLT**. The logical processor is inactive because it executed the HLT instruction.

  — 2: **Shutdown**. The logical processor is inactive because it incurred a **triple fault**[2] or some other serious error.

  — 3: **Wait-for-SIPI**. The logical processor is inactive because it is waiting for a startup-IPI (SIPI).

  Future processors may include support for other activity states. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine what activity states are supported.
- **Interruptibility state** (32 bits). The IA-32 architecture includes features that permit certain events to be blocked for a period of time. This field contains information about such blocking. Details and the format of this field are given in Table 26-3.

Table 26-3. Format of Interruptibility State

Bit Position(s)	Bit Name	Notes
0	Blocking by STI	See the "STI—Set Interrupt Flag" section in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B. Execution of STI with RFLAGS.IF = 0 blocks maskable interrupts on the instruction boundary following its execution.[1] Setting this bit indicates that this blocking is in effect.
1	Blocking by MOV SS	See Section 7.8.3, "Masking Exceptions and Interrupts When Switching Stacks," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. Execution of a MOV to SS or a POP to SS blocks or suppresses certain debug exceptions as well as interrupts (maskable and nonmaskable) on the instruction boundary following its execution. Setting this bit indicates that this blocking is in effect.[2] This document uses the term "blocking by MOV SS," but it applies equally to POP SS.
2	Blocking by SMI	See Section 33.2, "System Management Interrupt (SMI)." System-management interrupts (SMIs) are disabled while the processor is in system-management mode (SMM). Setting this bit indicates that blocking of SMIs is in effect.

1. Execution of the MWAIT instruction may put a logical processor into an inactive state. However, this VMCS field never reflects this state. See Section 29.1.
2. A triple fault occurs when a logical processor encounters an exception while attempting to deliver a double fault.

Table 26-3. Format of Interruptibility State (Contd.)

Bit Position(s)	Bit Name	Notes
3	Blocking by NMI	See Section 7.7.1, "Handling Multiple NMIs," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A and Section 33.8, "NMI Handling While in SMM." Delivery of a non-maskable interrupt (NMI) or a system-management interrupt (SMI) blocks subsequent NMIs until the next execution of IRET. See Section 27.3 for how this behavior of IRET may change in VMX non-root operation. Setting this bit indicates that blocking of NMIs is in effect. Clearing this bit does not imply that NMIs are not (temporarily) blocked for other reasons. If the "virtual NMIs" VM-execution control (see Section 26.6.1) is 1, this bit does not control the blocking of NMIs. Instead, it refers to "virtual-NMI blocking" (the fact that guest software is not ready for an NMI).
4	Enclave interruption	Set to 1 if the VM exit occurred while the logical processor was in enclave mode. Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode. A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified.
31:5	Reserved	VM entry will fail if these bits are not 0. See Section 28.3.1.5.

NOTES:

1. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.
2. System-management interrupts may also be inhibited on the instruction boundary following such an execution of MOV or POP.

- **Pending debug exceptions** (64 bits; 32 bits on processors that do not support Intel 64 architecture). IA-32 processors may recognize one or more debug exceptions without immediately delivering them.[1] This field contains information about such exceptions. This field is described in Table 26-4.

Table 26-4. Format of Pending-Debug-Exceptions

Bit Position(s)	Bit Name	Notes
3:0	B3 – B0	When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if the corresponding enabling bit in DR7 is not set.
10:4	Reserved	VM entry fails if these bits are not 0. See Section 28.3.1.5.
11	BLD	When set, this bit indicates that a bus lock was asserted while OS bus-lock detection was enabled and CPL > 0 (see Section 19.3.1.6, "OS Bus-Lock Detection").[1]
12	Enabled breakpoint	When set, this bit indicates that at least one data or I/O breakpoint was met and was enabled in DR7; the XBEGIN instruction was executed immediately before the VM exit and advanced debugging of RTM transactional regions had been enabled; or a bus lock was asserted while CPL > 0 and OS bus-lock detection had been enabled.
13	Reserved	VM entry fails if this bit is not 0. See Section 28.3.1.5.

1. For example, execution of a MOV to SS or a POP to SS may inhibit some debug exceptions for one instruction. See Section 7.8.3 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. In addition, certain events incident to an instruction (for example, an INIT signal) may take priority over debug traps generated by that instruction. See Table 7-2 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

Table 26-4. Format of Pending-Debug-Exceptions (Contd.)

Bit Position(s)	Bit Name	Notes
14	BS	When set, this bit indicates that a debug exception would have been triggered by single-step execution mode.
15	Reserved	VM entry fails if this bit is not 0. See Section 28.3.1.5.
16	RTM	When set, this bit indicates that a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 17.3.7, "RTM-Enabled Debugger Support," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).[2]
63:17	Reserved	VM entry fails if these bits are not 0. See Section 28.3.1.5. Bits 63:32 exist only on processors that support Intel 64 architecture.

NOTES:

1. In general, the format of this field matches that of DR6. However, DR6 **clears** bit 11 to indicate detection of a bus lock, while this field **sets** the bit to indicate that condition.
2. In general, the format of this field matches that of DR6. However, DR6 **clears** bit 16 to indicate an RTM-related exception, while this field **sets** the bit to indicate that condition.

- **VMCS link pointer** (64 bits). If the "VMCS shadowing" VM-execution control is 1, the VMREAD and VMWRITE instructions access the VMCS referenced by this pointer (see Section 26.10). Otherwise, software should set this field to FFFFFFFF_FFFFFFFFH to avoid VM-entry failures (see Section 28.3.1.5).
- **VMX-preemption timer value** (32 bits). This field is supported only on processors that support the 1-setting of the "activate VMX-preemption timer" VM-execution control. This field contains the value that the VMX-preemption timer will use following the next VM entry with that setting. See Section 27.5.1 and Section 28.7.4.
- **Page-directory-pointer-table entries** (PDPTEs; 64 bits each). These four (4) fields (PDPTE0, PDPTE1, PDPTE2, and PDPTE3) are supported only on processors that support the 1-setting of the "enable EPT" VM-execution control. They correspond to the PDPTEs referenced by CR3 when PAE paging is in use (see Section 5.4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). They are used only if the "enable EPT" VM-execution control is 1.
- **Guest interrupt status** (16 bits). This field is supported only on processors that support the 1-setting of the "virtual-interrupt delivery" VM-execution control. It characterizes part of the guest's virtual-APIC state and does not correspond to any processor or APIC registers. It comprises two 8-bit subfields:
  - **Requesting virtual interrupt (RVI)**. This is the low byte of the guest interrupt status. The processor treats this value as the vector of the highest priority virtual interrupt that is requesting service. (The value 0 implies that there is no such interrupt.)
  - **Servicing virtual interrupt (SVI)**. This is the high byte of the guest interrupt status. The processor treats this value as the vector of the highest priority virtual interrupt that is in service. (The value 0 implies that there is no such interrupt.)

  See Chapter 31 for more information on the use of this field.
- **PML index** (16 bits). This field is supported only on processors that support the 1-setting of the "enable PML" VM-execution control. It contains the logical index of the next entry in the page-modification log. Because the page-modification log comprises 512 entries, the PML index is typically a value in the range 0–511. Details of the page-modification log and use of the PML index are given in Section 30.3.6.

26.5 HOST-STATE AREA

This section describes fields contained in the host-state area of the VMCS. As noted earlier, processor state is loaded from these fields on every VM exit (see Section 29.5).

All fields in the host-state area correspond to processor registers:

- CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- RSP and RIP (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- Selector fields (16 bits each) for the segment registers CS, SS, DS, ES, FS, GS, and TR. There is no field in the host-state area for the LDTR selector.
- Base-address fields for FS, GS, TR, GDTR, and IDTR (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- The following MSRs:
  - IA32_SYSENTER_CS (32 bits)
  - IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture).
  - IA32_PERF_GLOBAL_CTRL (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-exit control.
  - IA32_PAT (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_PAT" VM-exit control.
  - IA32_EFER (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_EFER" VM-exit control.
  - IA32_S_CET (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-exit control.
  - IA32_INTERRUPT_SSP_TABLE_ADDR (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-exit control.
  - IA32_PKRS (64 bits). This field is supported only on processors that support the 1-setting of the "load PKRS" VM-exit control.
- The shadow-stack pointer register SSP (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-exit control.

In addition to the state identified here, some processor state components are loaded with fixed values on every VM exit; there are no fields corresponding to these components in the host-state area. See Section 29.5 for details of how state is loaded on VM exits.

26.6 VM-EXECUTION CONTROL FIELDS

The VM-execution control fields govern VMX non-root operation. These are described in Section 26.6.1 through Section 26.6.8.

26.6.1 Pin-Based VM-Execution Controls

The pin-based VM-execution controls constitute a 32-bit vector that governs the handling of asynchronous events (for example: interrupts).[1] Table 26-5 lists the controls. See Chapter 28 for how these controls affect processor behavior in VMX non-root operation.

1. Some asynchronous events cause VM exits regardless of the settings of the pin-based VM-execution controls (see Section 27.2).

Table 26-5. Definitions of Pin-Based VM-Execution Controls

Bit Position(s)	Name	Description
0	External-interrupt exiting	If this control is 1, external interrupts cause VM exits. Otherwise, they are delivered normally through the guest interrupt-descriptor table (IDT). If this control is 1, the value of RFLAGS.IF does not affect interrupt blocking.
3	NMI exiting	If this control is 1, non-maskable interrupts (NMIs) cause VM exits. Otherwise, they are delivered normally using descriptor 2 of the IDT. This control also determines interactions between IRET and blocking by NMI (see Section 27.3).
5	Virtual NMIs	If this control is 1, NMIs are never blocked and the "blocking by NMI" bit (bit 3) in the interruptibility-state field indicates "virtual-NMI blocking" (see Table 26-3). This control also interacts with the "NMI-window exiting" VM-execution control (see Section 26.6.2).
6	Activate VMX-preemption timer	If this control is 1, the VMX-preemption timer counts down in VMX non-root operation; see Section 27.5.1. A VM exit occurs when the timer counts down to zero; see Section 27.2.
7	Process posted interrupts	If this control is 1, the processor treats interrupts with the posted-interrupt notification vector (see Section 26.6.8) specially, updating the virtual-APIC page with posted-interrupt requests (see Section 31.6).

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_PINBASED_CTLS and IA32_VMX_TRUE_PINBASED_CTLS (see Appendix A.3.1) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 28.2.1.1).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 1, 2, and 4. The VMX capability MSR IA32_VMX_PINBASED_CTLS will always report that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_PINBASED_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

26.6.2 Processor-Based VM-Execution Controls

The processor-based VM-execution controls constitute three vectors that govern the handling of synchronous events, mainly those caused by the execution of specific instructions.[1] These are the **primary processor-based VM-execution controls** (32 bits), the **secondary processor-based VM-execution controls** (32 bits), and the tertiary **VM-execution controls** (64 bits).

Table 26-6 lists the primary processor-based VM-execution controls. See Chapter 26 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 26-6. Definitions of Primary Processor-Based VM-Execution Controls

Bit Position(s)	Name	Description
2	Interrupt-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if RFLAGS.IF = 1 and there are no other blocking of interrupts (see Section 26.4.2).
3	Use TSC offsetting	This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC offset field (see Section 26.6.5 and Section 27.3).
7	HLT exiting	This control determines whether executions of HLT cause VM exits.
9	INVLPG exiting	This determines whether executions of INVLPG and INVPCID cause VM exits.
10	MWAIT exiting	This control determines whether executions of MWAIT cause VM exits.
11	RDPMC exiting	This control determines whether executions of RDPMC cause VM exits.

1. Some instructions cause VM exits regardless of the settings of the processor-based VM-execution controls (see Section 27.1.2), as do task switches (see Section 27.2).

Table 26-6. Definitions of Primary Processor-Based VM-Execution Controls (Contd.)

Bit Position(s)	Name	Description
12	RDTSC exiting	This control determines whether executions of RDTSC and RDTSCP cause VM exits.
15	CR3-load exiting	In conjunction with the CR3-target controls (see Section 26.6.7), this control determines whether executions of MOV to CR3 cause VM exits. See Section 27.1.3. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
16	CR3-store exiting	This control determines whether executions of MOV from CR3 cause VM exits. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
17	Activate tertiary controls	This control determines whether the tertiary processor-based VM-execution controls are used. If this control is 0, the logical processor operates as if all the tertiary processor-based VM-execution controls were also 0.
19	CR8-load exiting	This control determines whether executions of MOV to CR8 cause VM exits.
20	CR8-store exiting	This control determines whether executions of MOV from CR8 cause VM exits.
21	Use TPR shadow	Setting this control to 1 enables TPR virtualization and other APIC-virtualization features. See Chapter 31.
22	NMI-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if there is no virtual-NMI blocking (see Section 26.4.2).
23	MOV-DR exiting	This control determines whether executions of MOV DR cause VM exits.
24	Unconditional I/O exiting	This control determines whether executions of I/O instructions (IN, INS/INSB/INSW/INSD, OUT, and OUTS/OUTSB/OUTSW/OUTSD) cause VM exits.
25	Use I/O bitmaps	This control determines whether I/O bitmaps are used to restrict executions of I/O instructions (see Section 26.6.4 and Section 27.1.3). For this control, "0" means "do not use I/O bitmaps" and "1" means "use I/O bitmaps." If the I/O bitmaps are used, the setting of the "unconditional I/O exiting" control is ignored.
27	Monitor trap flag	If this control is 1, the monitor trap flag debugging feature is enabled. See Section 27.5.2.
28	Use MSR bitmaps	This control determines whether MSR bitmaps are used to control execution of the RDMSR and WRMSR instructions (see Section 26.6.9 and Section 27.1.3). For this control, "0" means "do not use MSR bitmaps" and "1" means "use MSR bitmaps." If the MSR bitmaps are not used, all executions of the RDMSR and WRMSR instructions cause VM exits.
29	MONITOR exiting	This control determines whether executions of MONITOR cause VM exits.
30	PAUSE exiting	This control determines whether executions of PAUSE cause VM exits.
31	Activate secondary controls	This control determines whether the secondary processor-based VM-execution controls are used. If this control is 0, the logical processor operates as if all the secondary processor-based VM-execution controls were also 0.

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_PROCBASED_CTLS and IA32_VMX_TRUE_PROCBASED_CTLS (see Appendix A.3.2) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 28.2.1.1).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 1, 4–6, 8, 13–16, and 26. The VMX capability MSR IA32_VMX_PROCBASED_CTLS will always report that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_PROCBASED_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

Bit 31 of the primary processor-based VM-execution controls determines whether the secondary processor-based VM-execution controls are used. If that bit is 0, VM entry and VMX non-root operation function as if all the secondary processor-based VM-execution controls were 0. Processors that support only the 0-setting of bit 31 of the primary processor-based VM-execution controls do not support the secondary processor-based VM-execution controls.

Table 26-7 lists the secondary processor-based VM-execution controls. See Chapter 26 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 26-7. Definitions of Secondary Processor-Based VM-Execution Controls

Bit Position(s)	Name	Description
0	Virtualize APIC accesses	If this control is 1, the logical processor treats specially accesses to the page with the APIC-access address. See Section 31.4.
1	Enable EPT	If this control is 1, extended page tables (EPT) are enabled. See Section 30.3.
2	Descriptor-table exiting	This control determines whether executions of LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, and STR cause VM exits.
3	Enable RDTSCP	If this control is 0, any execution of RDTSCP causes an invalid-opcode exception (#UD).
4	Virtualize x2APIC mode	If this control is 1, the logical processor treats specially RDMSR and WRMSR to APIC MSRs (in the range 800H–8FFH). See Section 31.5.
5	Enable VPID	If this control is 1, cached translations of linear addresses are associated with a virtual-processor identifier (VPID). See Section 30.1.
6	WBINVD exiting	This control determines whether executions of WBINVD and WBNOINVD cause VM exits.
7	Unrestricted guest	This control determines whether guest software may run in unpaged protected mode or in real-address mode.
8	APIC-register virtualization	If this control is 1, the logical processor virtualizes certain APIC accesses. See Section 31.4 and Section 31.5.
9	Virtual-interrupt delivery	This controls enables the evaluation and delivery of pending virtual interrupts as well as the emulation of writes to the APIC registers that control interrupt prioritization.
10	PAUSE-loop exiting	This control determines whether a series of executions of PAUSE can cause a VM exit (see Section 26.6.13 and Section 27.1.3).
11	RDRAND exiting	This control determines whether executions of RDRAND cause VM exits.
12	Enable INVPCID	If this control is 0, any execution of INVPCID causes a #UD.
13	Enable VM functions	Setting this control to 1 enables use of the VMFUNC instruction in VMX non-root operation. See Section 27.5.6.
14	VMCS shadowing	If this control is 1, executions of VMREAD and VMWRITE in VMX non-root operation may access a shadow VMCS (instead of causing VM exits). See Section 26.10 and Section 32.3.
15	Enable ENCLS exiting	If this control is 1, executions of ENCLS consult the ENCLS-exiting bitmap to determine whether the instruction causes a VM exit. See Section 26.6.16 and Section 27.1.3.
16	RDSEED exiting	This control determines whether executions of RDSEED cause VM exits.
17	Enable PML	If this control is 1, an access to a guest-physical address that sets an EPT dirty bit first adds an entry to the page-modification log. See Section 30.3.6.
18	EPT-violation #VE	If this control is 1, EPT violations may cause virtualization exceptions (#VE) instead of VM exits. See Section 27.5.7.
19	Conceal VMX from PT	If this control is 1, Intel Processor Trace suppresses from PIPs an indication that the processor was in VMX non-root operation and omits a VMCS packet from any PSB+ produced in VMX non-root operation (see Chapter 34).
20	Enable XSAVES/XRSTORS	If this control is 0, any execution of XSAVES or XRSTORS causes a #UD.
21	PASID translation	If this control is 1, PASID translation is performed for executions of ENQCMD and ENQCMDS. See Section 27.5.8.
22	Mode-based execute control for EPT	If this control is 1, EPT execute permissions are based on whether the linear address being accessed is supervisor mode or user mode. See Chapter 30.

Table 26-7. Definitions of Secondary Processor-Based VM-Execution Controls (Contd.)

Bit Position(s)	Name	Description
23	Sub-page write permissions for EPT	If this control is 1, EPT write permissions may be specified at the granularity of 128 bytes. See Section 30.3.4.
24	Intel PT uses guest physical addresses	If this control is 1, all output addresses used by Intel Processor Trace are treated as guest-physical addresses and translated using EPT. See Section 27.5.4.
25	Use TSC scaling	This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC multiplier field (see Section 26.6.5 and Section 27.3).
26	Enable user wait and pause	If this control is 0, any execution of TPAUSE, UMONITOR, or UMWAIT causes a #UD.
27	Enable PCONFIG	If this control is 0, any execution of PCONFIG causes a #UD.
28	Enable ENCLV exiting	If this control is 1, executions of ENCLV consult the ENCLV-exiting bitmap to determine whether the instruction causes a VM exit. See Section 26.6.17 and Section 27.1.3.
30	VMM bus-lock detection	This control determines whether assertion of a bus lock causes a VM exit. See Section 27.2.
31	Instruction timeout	If this control is 1, a VM exit occurs if certain operations prevent the processor from reaching an instruction boundary within a specified amount of time. See Section 26.6.25 and Section 27.2.

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_PROCBASED_CTLS2 (see Appendix A.3.3) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 28.2.1.1).

Bit 17 of the primary processor-based VM-execution controls determines whether the tertiary processor-based VM-execution controls are used. If that bit is 0, VM entry and VMX non-root operation function as if all the tertiary processor-based VM-execution controls were 0. Processors that support only the 0-setting of bit 17 of the primary processor-based VM-execution controls do not support the tertiary processor-based VM-execution controls.

Table 26-8 lists the tertiary processor-based VM-execution controls. See Chapter 26 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 26-8. Definitions of Tertiary Processor-Based VM-Execution Controls

Bit Position(s)	Name	Description
0	LOADIWKEY exiting	This control determines whether executions of LOADIWKEY cause VM exits.
1	Enable HLAT	This control enables hypervisor-managed linear-address translation. See Section 5.5.1.
2	EPT paging-write control	If this control is 1, EPT permissions can be specified to allow writes only for paging-related updates. See Section 30.3.3.2.
3	Guest-paging verification	If this control is 1, EPT permissions can be specified to prevent accesses using linear addresses whose translation has certain properties. See Section 30.3.3.2.
4	IPI virtualization	If this control is 1, virtualization of interprocessor interrupts (IPIs) is enabled. See Section 31.1.6.
6	Enable MSR-list instructions	If this control is 0, any execution of RDMSRLIST or WRMSRLIST causes a #UD.
7	Virtualize IA32_SPEC_CTRL	If this control is 1, the operation of the RDMSR and WRMSR instructions is changed when accessing the IA32_SPEC_CTRL MSR. See Section 26.3.

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_PROCBASED_CTLS3 (see Appendix A.3.4) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 28.2.1.1).

26.6.3 Exception Bitmap

The **exception bitmap** is a 32-bit field that contains one bit for each exception. When an exception occurs, its vector is used to select a bit in this field. If the bit is 1, the exception causes a VM exit. If the bit is 0, the exception is delivered normally through the IDT, using the descriptor corresponding to the exception's vector.

Whether a page fault (exception with vector 14) causes a VM exit is determined by bit 14 in the exception bitmap as well as the error code produced by the page fault and two 32-bit fields in the VMCS (the **page-fault error-code mask** and **page-fault error-code match)**. See Section 27.2 for details.

26.6.4 I/O-Bitmap Addresses

The VM-execution control fields include the 64-bit physical addresses of **I/O bitmaps** A and B (each of which are 4 KBytes in size). I/O bitmap A contains one bit for each I/O port in the range 0000H through 7FFFH; I/O bitmap B contains bits for ports in the range 8000H through FFFFH.

A logical processor uses these bitmaps if and only if the "use I/O bitmaps" control is 1. If the bitmaps are used, execution of an I/O instruction causes a VM exit if any bit in the I/O bitmaps corresponding to a port it accesses is 1. See Section 27.1.3 for details. If the bitmaps are used, their addresses must be 4-KByte aligned.

26.6.5 Time-Stamp Counter Offset and Multiplier

The VM-execution control fields include a 64-bit **TSC-offset** field. If the "RDTSC exiting" control is 0 and the "use TSC offsetting" control is 1, this field controls executions of the RDTSC and RDTSCP instructions. It also controls executions of the RDMSR instruction that read from the IA32_TIME_STAMP_COUNTER MSR. For all of these, the value of the TSC offset is added to the value of the time-stamp counter, and the sum is returned to guest software in EDX:EAX.

Processors that support the 1-setting of the "use TSC scaling" control also support a 64-bit **TSC-multiplier** field. If this control is 1 (and the "RDTSC exiting" control is 0 and the "use TSC offsetting" control is 1), this field also affects the executions of the RDTSC, RDTSCP, and RDMSR instructions identified above. Specifically, the contents of the time-stamp counter is first multiplied by the TSC multiplier before adding the TSC offset.

See Chapter 26 for a detailed treatment of the behavior of RDTSC, RDTSCP, and RDMSR in VMX non-root operation.

26.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4

VM-execution control fields include **guest/host masks** and **read shadows** for the CR0 and CR4 registers. These fields control executions of instructions that access those registers (including CLTS, LMSW, MOV CR, and SMSW). They are 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not.

In general, bits set to 1 in a guest/host mask correspond to bits "owned" by the host:

- Guest attempts to set them (using CLTS, LMSW, or MOV to CR) to values differing from the corresponding bits in the corresponding read shadow cause VM exits.
- Guest reads (using MOV from CR or SMSW) return values for these bits from the corresponding read shadow.

Bits cleared to 0 correspond to bits "owned" by the guest; guest attempts to modify them succeed and guest reads return values for these bits from the control register itself.

See Chapter 28 for details regarding how these fields affect VMX non-root operation.

26.6.7 CR3-Target Controls

The VM-execution control fields include a set of 4 **CR3-target values** and a **CR3-target count**. The CR3-target values each have 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not. The CR3-target count has 32 bits on all processors.

An execution of MOV to CR3 in VMX non-root operation does not cause a VM exit if its source operand matches one of these values. If the CR3-target count is *n*, only the first *n* CR3-target values are considered; if the CR3-target count is 0, MOV to CR3 always causes a VM exit.

There are no limitations on the values that can be written for the CR3-target values. VM entry fails (see Section 28.2) if the CR3-target count is greater than 4.

Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine the number of values supported.

26.6.8 Controls for APIC Virtualization

There are three mechanisms by which software accesses registers of the logical processor's local APIC:

- If the local APIC is in xAPIC mode, it can perform memory-mapped accesses to addresses in the 4-KByte page referenced by the physical address in the IA32_APIC_BASE MSR (see Section 12.4.4, "Local APIC Status and Location," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, and the Intel® 64 Architecture Processor Topology Enumeration Technical Paper).[1]
- If the local APIC is in x2APIC mode, it can accesses the local APIC's registers using the RDMSR and WRMSR instructions (see the Intel® 64 Architecture Processor Topology Enumeration Technical Paper).
- In 64-bit mode, it can access the local APIC's task-priority register (TPR) using the MOV CR8 instruction.

Several processor-based VM-execution controls (see Section 26.6.2) control such accesses. These are "use TPR shadow", "virtualize APIC accesses", "virtualize x2APIC mode", "virtual-interrupt delivery", "APIC-register virtualization", and "IPI virtualization". These controls interact with the following fields:

- **APIC-access address** (64 bits). This field contains the physical address of the 4-KByte **APIC-access page**. If the "virtualize APIC accesses" VM-execution control is 1, access to this page may cause VM exits or be virtualized by the processor. See Section 31.4.

  The APIC-access address exists only on processors that support the 1-setting of the "virtualize APIC accesses" VM-execution control.
- **Virtual-APIC address** (64 bits). This field contains the physical address of the 4-KByte **virtual-APIC page**. The processor uses the virtual-APIC page to virtualize certain accesses to APIC registers and to manage virtual interrupts; see Chapter 31.

  Depending on the setting of the controls indicated earlier, the virtual-APIC page may be accessed by the following operations:

  — The MOV CR8 instructions (see Section 31.3).

  — Accesses to the APIC-access page if, in addition, the "virtualize APIC accesses" VM-execution control is 1 (see Section 31.4).

  — The RDMSR and WRMSR instructions if, in addition, the value of ECX is in the range 800H–8FFH (indicating an APIC MSR) and the "virtualize x2APIC mode" VM-execution control is 1 (see Section 31.5).

  If the "use TPR shadow" VM-execution control is 1, VM entry ensures that the virtual-APIC address is 4-KByte aligned. The virtual-APIC address exists only on processors that support the 1-setting of the "use TPR shadow" VM-execution control.
- **TPR threshold** (32 bits). Bits 3:0 of this field determine the threshold below which bits 7:4 of VTPR (see Section 31.1.1) cannot fall. If the "virtual-interrupt delivery" VM-execution control is 0, a VM exit occurs after an operation (e.g., an execution of MOV to CR8) that reduces the value of those bits below the TPR threshold. See Section 31.1.2.

  The TPR threshold exists only on processors that support the 1-setting of the "use TPR shadow" VM-execution control.
- **EOI-exit bitmap** (4 fields; 64 bits each). These fields are supported only on processors that support the 1-setting of the "virtual-interrupt delivery" VM-execution control. They are used to determine which virtualized writes to the APIC's EOI register cause VM exits:

1. If the local APIC does not support x2APIC mode, it is always in xAPIC mode.

  - EOI_EXIT0 contains bits for vectors from 0 (bit 0) to 63 (bit 63).
  - EOI_EXIT1 contains bits for vectors from 64 (bit 0) to 127 (bit 63).
  - EOI_EXIT2 contains bits for vectors from 128 (bit 0) to 191 (bit 63).
  - EOI_EXIT3 contains bits for vectors from 192 (bit 0) to 255 (bit 63).

  See Section 31.1.4 for more information on the use of this field.

- **Posted-interrupt notification vector** (16 bits). This field is supported only on processors that support the 1-setting of the "process posted interrupts" VM-execution control. Its low 8 bits contain the interrupt vector that is used to notify a logical processor that virtual interrupts have been posted. See Section 31.6 for more information on the use of this field.
- **Posted-interrupt descriptor address** (64 bits). This field is supported only on processors that support the 1-setting of the "process posted interrupts" VM-execution control. It is the physical address of a 64-byte aligned posted interrupt descriptor. See Section 31.6 for more information on the use of this field.
- **PID-pointer table address** (64 bits). This field contains the physical address of the **PID-pointer table**. If the "IPI virtualization" VM-execution control is 1, the logical processor uses entries in this table to virtualize IPIs. See Section 31.1.6.
- **Last PID-pointer index** (16 bits). This field contains the index of the last entry in the PID-pointer table.

26.6.9 MSR-Bitmap Address

On processors that support the 1-setting of the "use MSR bitmaps" VM-execution control, the VM-execution control fields include the 64-bit physical address of four contiguous **MSR bitmaps**, which are each 1-KByte in size. This field does not exist on processors that do not support the 1-setting of that control. The four bitmaps are:

- **Read bitmap for low MSRs** (located at the MSR-bitmap address)**.** This contains one bit for each MSR address in the range 00000000H to 00001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- **Read bitmap for high MSRs** (located at the MSR-bitmap address plus 1024)**.** This contains one bit for each MSR address in the range C0000000H toC0001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- **Write bitmap for low MSRs** (located at the MSR-bitmap address plus 2048)**.** This contains one bit for each MSR address in the range 00000000H to 00001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.
- **Write bitmap for high MSRs** (located at the MSR-bitmap address plus 3072)**.** This contains one bit for each MSR address in the range C0000000H toC0001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

A logical processor uses these bitmaps if and only if the "use MSR bitmaps" control is 1. If the bitmaps are used, an execution of RDMSR or WRMSR causes a VM exit if the value of RCX is in neither of the ranges covered by the bitmaps or if the appropriate bit in the MSR bitmaps (corresponding to the instruction and the RCX value) is 1. See Section 27.1.3 for details. If the bitmaps are used, their address must be 4-KByte aligned.

26.6.10 Executive-VMCS Pointer

The executive-VMCS pointer is a 64-bit field used in the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). SMM VM exits save this field as described in Section 33.15.2. VM entries that return from SMM use this field as described in Section 33.15.4.

26.6.11 Extended-Page-Table Pointer (EPTP)

The **extended-page-table pointer** (EPTP) contains the address of the base of EPT PML4 table (see Section 30.3.2), as well as other EPT configuration information. The format of this field is shown in Table 26-9.

Table 26-9. Format of Extended-Page-Table Pointer

Bit Position(s)	Field
2:0	EPT paging-structure memory type (see Section 30.3.7): 0 = Uncacheable (UC) 6 = Write-back (WB) Other values are reserved.[1]
5:3	This value is 1 less than the EPT page-walk length (see Section 30.3.2)
6	Setting this control to 1 enables accessed and dirty flags for EPT (see Section 30.3.5)[2]
7	Setting this control to 1 enables enforcement of access rights for supervisor shadow-stack pages (see Section 30.3.3.2)[3]
11:8	Reserved
N–1:12	Bits N–1:12 of the physical address of the 4-KByte aligned EPT paging-structure (an EPT PML4 table with 4-level EPT and an EPT PML5 table with 5-level EPT)[4]
63:N	Reserved

NOTES:

1. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine what EPT paging-structure memory types are supported.
2. Not all processors support accessed and dirty flags for EPT. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_-CAP (see Appendix A.10) to determine whether the processor supports this feature.
3. Not all processors enforce access rights for shadow-stack pages. Software should read the VMX capability MSR IA32_VMX-_EPT_VPID_CAP (see Appendix A.10) to determine whether the processor supports this feature.
4. N is the physical-address width supported by the logical processor. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

The EPTP exists only on processors that support the 1-setting of the "enable EPT" VM-execution control.

26.6.12 Virtual-Processor Identifier (VPID)

The **virtual-processor identifier** (VPID) is a 16-bit field. It exists only on processors that support the 1-setting of the "enable VPID" VM-execution control. See Section 30.1 for details regarding the use of this field.

26.6.13 Controls for PAUSE-Loop Exiting

On processors that support the 1-setting of the "PAUSE-loop exiting" VM-execution control, the VM-execution control fields include the following 32-bit fields:

- **PLE_Gap**. Software can configure this field as an upper bound on the amount of time between two successive executions of PAUSE in a loop.
- **PLE_Window**. Software can configure this field as an upper bound on the amount of time a guest is allowed to execute in a PAUSE loop.

These fields measure time based on a counter that runs at the same rate as the timestamp counter (TSC). See Section 27.1.3 for more details regarding PAUSE-loop exiting.

26.6.14 VM-Function Controls

The **VM-function controls** constitute a 64-bit vector that governs use of the VMFUNC instruction in VMX non-root operation. This field is supported only on processors that support the 1-settings of both the "activate secondary controls" primary processor-based VM-execution control and the "enable VM functions" secondary processor-based VM-execution control.

Table 26-10 lists the VM-function controls. See Section 27.5.6 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 26-10. Definitions of VM-Function Controls

Bit Position(s)	Name	Description
0	EPTP switching	The EPTP-switching VM function changes the EPT pointer to a value chosen from the EPTP list. See Section 27.5.6.3.

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_VMFUNC (see Appendix A.11) to determine which bits are reserved. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 28.2.1.1).

Processors that support the 1-setting of the "EPTP switching" VM-function control also support a 64-bit field called the **EPTP-list address**. This field contains the physical address of the 4-KByte **EPTP list**. The EPTP list comprises 512 8-Byte entries (each an EPTP value) and is used by the EPTP-switching VM function (see Section 27.5.6.3).

26.6.15 VMCS Shadowing Bitmap Addresses

On processors that support the 1-setting of the "VMCS shadowing" VM-execution control, the VM-execution control fields include the 64-bit physical addresses of the **VMREAD bitmap** and the **VMWRITE bitmap**. Each bitmap is 4 KBytes in size and thus contains 32 KBits. The addresses are the **VMREAD-bitmap address** and the **VMWRITE-bitmap address**.

If the "VMCS shadowing" VM-execution control is 1, executions of VMREAD and VMWRITE may consult these bitmaps (see Section 26.10 and Section 32.3).

26.6.16 ENCLS-Exiting Bitmap

The **ENCLS-exiting bitmap** is a 64-bit field. If the "enable ENCLS exiting" VM-execution control is 1, execution of ENCLS causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the bit is 0, the instruction executes normally. See Section 27.1.3 for more information.

26.6.17 ENCLV-Exiting Bitmap

The **ENCLV-exiting bitmap** is a 64-bit field. If the "enable ENCLV exiting" VM-execution control is 1, execution of ENCLV causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the bit is 0, the instruction executes normally. See Section 27.1.3 for more information.

26.6.18 PCONFIG-Exiting Bitmap

The **PCONFIG-exiting bitmap** is a 64-bit field. If the "enable PCONFIG" VM-execution control is 1, execution of PCONFIG causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the control is 0, any execution of PCONFIG causes a #UD. See Section 27.1.3 for more information.

26.6.19 Control Field for Page-Modification Logging

The **PML address** is a 64-bit field. It is the 4-KByte aligned address of the **page-modification log**. The page-modification log consists of 512 64-bit entries. It is used for the page-modification logging feature. Details of the page-modification logging are given in Section 30.3.6.

If the "enable PML" VM-execution control is 1, VM entry ensures that the PML address is 4-KByte aligned. The PML address exists only on processors that support the 1-setting of the "enable PML" VM-execution control.

26.6.20 Controls for Virtualization Exceptions

On processors that support the 1-setting of the "EPT-violation #VE" VM-execution control, the VM-execution control fields include the following:

- **Virtualization-exception information address** (64 bits). This field contains the physical address of the **virtualization-exception information area**. When a logical processor encounters a virtualization exception, it saves virtualization-exception information at the virtualization-exception information address; see Section 27.5.7.2.
- **EPTP index** (16 bits). When an EPT violation causes a virtualization exception, the processor writes the value of this field to the virtualization-exception information area. The EPTP-switching VM function updates this field (see Section 27.5.6.3).

26.6.21 XSS-Exiting Bitmap

On processors that support the 1-setting of the "enable XSAVES/XRSTORS" VM-execution control, the VM-execution control fields include a 64-bit **XSS-exiting bitmap**. If the "enable XSAVES/XRSTORS" VM-execution control is 1, executions of XSAVES and XRSTORS may consult this bitmap (see Section 27.1.3 and Section 27.3).

26.6.22 Sub-Page-Permission-Table Pointer (SPPTP)

If the sub-page write-permission feature of EPT is enabled, EPT write permissions may be determined at a 128-byte granularity (see Section 30.3.4). These permissions are determined using a hierarchy of sub-page-permission structures in memory.

The root of this hierarchy is referenced by a VM-execution control field called the **sub-page-permission-table pointer** (SPPTP). The SPPTP contains the address of the base of the root SPP table (see Section 30.3.4.2). The format of this field is shown in Table 26-9.

Table 26-11. Format of Sub-Page-Permission-Table Pointer

Bit Position(s)	Field
11:0	Reserved
N–1:12	Bits N–1:12 of the physical address of the 4-KByte aligned root SPP table
63:N[1]	Reserved

NOTES:

1. N is the processor's physical-address width. Software can determine this width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

The SPPTP exists only on processors that support the 1-setting of the "sub-page write permissions for EPT" VM-execution control.

26.6.23 Fields Related to Hypervisor-Managed Linear-Address Translation

Two fields are used when the "enable HLAT" VM-execution control is 1, enabling HLAT paging:

- The **hypervisor-managed linear-address translation pointer** (HLAT pointer or HLATP) is used by HLAT paging to locate and access the first paging structure used for linear-address translation (see Section 5.5). The format of this field is shown in Table 26-12.

Table 26-12. Format of Hypervisor-Managed Linear-Address Translation Pointer

Bit Position(s)	Field
2:0	Reserved
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the first HLAT paging structure during linear-address translation.
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the first HLAT paging structure during linear-address translation.
11:5	Reserved
N–1:12	Guest-physical address (4KB-aligned) of the first HLAT paging structure during linear-address translation.[1]
63:N	Reserved

NOTES:

1. N is the physical-address width supported by the logical processor. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- The HLAT prefix size. The value of this field determines which linear address are subject to HLAT paging. See Section 5.5.1.

These fields exist only on processors that support the 1-setting of the "enable HLAT" VM-execution control.

26.6.24 Fields Related to PASID Translation

Two 64-bit VM-execution control fields are used when the "PASID translation" VM-execution control is 1, enabling translation of PASIDs for executions of ENQCMD and ENQCMDS: the **low PASID directory address** and the **high PASID directory address**. These are the physical addresses of the low PASID directory and the high PASID directory, respectively. These fields exist only on processors that support the 1-setting of the "PASID translation" VM-execution control.

See Section 27.5.8 for information on the PASID-translation process for ENQCMD and ENQCMDS.

26.6.25 Instruction-Timeout Control

On processors that support the 1-setting of the "instruction timeout" VM-execution control, the VM-execution control fields include a 32-bit **instruction-timeout control**. The processor interprets the value of this field as an amount of time as measured in units of crystal clock cycles.[1] If the "instruction timeout" VM-execution control is 1, a VM exit occurs if certain operations prevent the processor from reaching an instruction boundary within this amount of time.

1. CPUID.15H:ECX enumerates the nominal frequency of the core crystal clock in Hz.

26.6.26 Fields Controlling Virtualization of the IA32_SPEC_CTRL MSR

On processors that support the 1-setting of the "virtualize IA32_SPEC_CTRL" VM-execution control, the VM-execution control fields include the following 64-bit fields:

- **IA32_SPEC_CTRL mask.** Setting a bit in this field prevents guest software from modifying the corresponding bit in the IA32_SPEC_CTRL MSR.
- **IA32_SPEC_CTRL shadow.** This field contains the value that guest software expects to be in the IA32_SPEC_CTRL MSR.

Section 27.3 discusses how these fields are used in VMX non-root operation.

26.7 VM-EXIT CONTROL FIELDS

The VM-exit control fields govern the behavior of VM exits. They are discussed in Section 26.7.1 and Section 26.7.2.

26.7.1 VM-Exit Controls

The VM-exit controls constitute two vectors that govern the basic operation of VM exits. These are the **primary VM-exit controls** (32 bits) and the **secondary VM-exits controls** (64 bits).

Table 26-13 lists the primary VM-exit controls. See Chapter 28 for complete details of how these controls affect VM exits.

Table 26-13. Definitions of Primary VM-Exit Controls

Bit Position(s)	Name	Description
2	Save debug controls	This control determines whether DR7 and the IA32_DEBUGCTL MSR are saved on VM exit. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
9	Host address-space size	On processors that support Intel 64 architecture, this control determines whether a logical processor is in 64-bit mode after the next VM exit. Its value is loaded into CS.L, IA32_EFER.LME, and IA32_EFER.LMA on every VM exit.[1] This control must be 0 on processors that do not support Intel 64 architecture.
12	Load IA32_PERF_GLOBAL_CTRL	This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM exit.
15	Acknowledge interrupt on exit	This control affects VM exits due to external interrupts: ▪ If such a VM exit occurs and this control is 1, the logical processor acknowledges the interrupt controller, acquiring the interrupt's vector. The vector is stored in the VM-exit interruption-information field, which is marked valid. ▪ If such a VM exit occurs and this control is 0, the interrupt is not acknowledged and the VM-exit interruption-information field is marked invalid.
18	Save IA32_PAT	This control determines whether the IA32_PAT MSR is saved on VM exit.
19	Load IA32_PAT	This control determines whether the IA32_PAT MSR is loaded on VM exit.
20	Save IA32_EFER	This control determines whether the IA32_EFER MSR is saved on VM exit.
21	Load IA32_EFER	This control determines whether the IA32_EFER MSR is loaded on VM exit.
22	Save VMX-preemption timer value	This control determines whether the value of the VMX-preemption timer is saved on VM exit.
23	Clear IA32_BNDCFGS	This control determines whether the IA32_BNDCFGS MSR is cleared on VM exit.
24	Conceal VMX from PT	If this control is 1, Intel Processor Trace does not produce a paging information packet (PIP) on a VM exit or a VMCS packet on an SMM VM exit (see Chapter 34).

Table 26-13. Definitions of Primary VM-Exit Controls (Contd.)

Bit Position(s)	Name	Description
25	Clear IA32_RTIT_CTL	This control determines whether the IA32_RTIT_CTL MSR is cleared on VM exit.
26	Clear IA32_LBR_CTL	This control determines whether the IA32_LBR_CTL MSR is cleared on VM exit.
27	Clear UINV	This control determines whether UINV is cleared on VM exit.
28	Load CET state	This control determines whether CET-related MSRs and SSP are loaded on VM exit.
29	Load PKRS	This control determines whether the IA32_PKRS MSR is loaded on VM exit.
30	Save IA32_PERF_GLOBAL_CTL	This control determines whether the IA32_PERF_GLOBAL_CTL MSR is saved on VM exit.
31	Activate secondary controls	This control determines whether the secondary VM-exit controls are used. If this control is 0, the logical processor operates as if all the secondary VM-exit controls were also 0.

NOTES:

1. Since the Intel 64 architecture specifies that IA32_EFER.LMA is always set to the logical-AND of CR0.PG and IA32_EFER.LME, and since CR0.PG is always 1 in VMX root operation, IA32_EFER.LMA is always identical to IA32_EFER.LME in VMX root operation.

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_EXIT_CTLS and IA32_VMX_TRUE_EXIT_CTLS (see Appendix A.4) to determine how it should set the reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 28.2.1.2).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 0–8, 10, 11, 13, 14, 16, and 17. The VMX capability MSR IA32_VMX_EXIT_CTLS always reports that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_EXIT_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

Bit 31 of the primary processor-based VM-exit controls determines whether the secondary VM-exit controls are used. If that bit is 0, VM entries and VM exits function as if all the secondary VM-exit controls were 0. Processors that support only the 0-setting of bit 31 of the primary VM-exit controls do not support the secondary VM-exit controls.

Table 26-14 lists the secondary VM-exit controls. See Chapter 28 for more details of how these controls affect VM exits.

Table 26-14. Definitions of Secondary VM-Exit Controls

Bit Position(s)	Name	Description
3	Prematurely busy shadow stack	If this control is 1, VM exits that cause a shadow stack to become prematurely busy (see Section 18.2.3, “Supervisor Shadow Stack Token,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) indicate this fact and save additional information into the VMCS.

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_EXIT_CTLS2 (see Appendix A.4.2) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 28.2.1.2).

26.7.2 VM-Exit Controls for MSRs

A VMM may specify lists of MSRs to be stored and loaded on VM exits. The following VM-exit control fields determine how MSRs are stored on VM exits:

- **VM-exit MSR-store count** (32 bits). This field specifies the number of MSRs to be stored on VM exit. It is recommended that this count not exceed 512.[1] Otherwise, unpredictable processor behavior (including a machine check) may result during VM exit.

- **VM-exit MSR-store address** (64 bits). This field contains the physical address of the VM-exit MSR-store area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-store count. The format of each entry is given in Table 26-15. If the VM-exit MSR-store count is not zero, the address must be 16-byte aligned.

Table 26-15. Format of an MSR Entry

Bit Position(s)	Contents
31:0	MSR index
63:32	Reserved
127:64	MSR data

See Section 29.4 for how this area is used on VM exits.

The following VM-exit control fields determine how MSRs are loaded on VM exits:

- **VM-exit MSR-load count** (32 bits). This field contains the number of MSRs to be loaded on VM exit. It is recommended that this count not exceed 512. Otherwise, unpredictable processor behavior (including a machine check) may result during VM exit.[1]
- **VM-exit MSR-load address** (64 bits). This field contains the physical address of the VM-exit MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-load count (see Table 26-15). If the VM-exit MSR-load count is not zero, the address must be 16-byte aligned.

See Section 29.6 for how this area is used on VM exits.

26.8 VM-ENTRY CONTROL FIELDS

The VM-entry control fields govern the behavior of VM entries. They are discussed in Sections 26.8.1 through 26.8.3.

26.8.1 VM-Entry Controls

The **VM-entry controls** constitute a 32-bit vector that governs the basic operation of VM entries. Table 26-16 lists the controls supported. See Chapter 26 for how these controls affect VM entries.

Table 26-16. Definitions of VM-Entry Controls

Bit Position(s)	Name	Description
2	Load debug controls	This control determines whether DR7 and the IA32_DEBUGCTL MSR are loaded on VM entry. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
9	IA-32e mode guest	On processors that support Intel 64 architecture, this control determines whether the logical processor is in IA-32e mode after VM entry. Its value is loaded into IA32_EFER.LMA as part of VM entry.[1] This control must be 0 on processors that do not support Intel 64 architecture.
10	Entry to SMM	This control determines whether the logical processor is in system-management mode (SMM) after VM entry. This control must be 0 for any VM entry from outside SMM.
11	Deactivate dual-monitor treatment	If set to 1, the default treatment of SMIs and SMM is in effect after the VM entry (see Section 33.15.7). This control must be 0 for any VM entry from outside SMM.

1. Future implementations may allow more MSRs to be stored reliably. Software should consult the VMX capability MSR IA32_VMX_-MISC to determine the number supported (see Appendix A.6).

1. Future implementations may allow more MSRs to be loaded reliably. Software should consult the VMX capability MSR IA32_VMX_-MISC to determine the number supported (see Appendix A.6).

Table 26-16. Definitions of VM-Entry Controls (Contd.)

Bit Position(s)	Name	Description
13	Load IA32_PERF_GLOBAL_CTRL	This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM entry.
14	Load IA32_PAT	This control determines whether the IA32_PAT MSR is loaded on VM entry.
15	Load IA32_EFER	This control determines whether the IA32_EFER MSR is loaded on VM entry.
16	Load IA32_BNDCFGS	This control determines whether the IA32_BNDCFGS MSR is loaded on VM entry.
17	Conceal VMX from PT	If this control is 1, Intel Processor Trace does not produce a paging information packet (PIP) on a VM entry or a VMCS packet on a VM entry that returns from SMM (see Chapter 34).
18	Load IA32_RTIT_CTL	This control determines whether the IA32_RTIT_CTL MSR is loaded on VM entry.
19	Load UINV	This control determines whether UINV is loaded on VM entry.
20	Load CET state	This control determines whether CET-related MSRs and SSP are loaded on VM entry.
21	Load guest IA32_LBR_CTL	This control determines whether the IA32_LBR_CTL MSR is loaded on VM entry.
22	Load PKRS	This control determines whether the IA32_PKRS MSR is loaded on VM entry.

NOTES:

1. Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control. If it is read as 1, every VM exit stores the value of IA32_EFER.LMA into the “IA-32e mode guest” VM-entry control (see Section 29.2).

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_ENTRY_CTLS and IA32_VMX_TRUE_ENTRY_CTLS (see Appendix A.5) to determine how it should set the reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 28.2.1.3).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 0–8 and 12. The VMX capability MSR IA32_VMX_ENTRY_CTLS always reports that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_ENTRY_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

26.8.2 VM-Entry Controls for MSRs

A VMM may specify a list of MSRs to be loaded on VM entries. The following VM-entry control fields manage this functionality:

- **VM-entry MSR-load count** (32 bits). This field contains the number of MSRs to be loaded on VM entry. It is recommended that this count not exceed 512. Otherwise, unpredictable processor behavior (including a machine check) may result during VM entry.[1]
- **VM-entry MSR-load address** (64 bits). This field contains the physical address of the VM-entry MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-entry MSR-load count. The format of entries is described in Table 26-15. If the VM-entry MSR-load count is not zero, the address must be 16-byte aligned.

See Section 28.4 for details of how this area is used on VM entries.

26.8.3 VM-Entry Controls for Event Injection

1. Future implementations may allow more MSRs to be loaded reliably. Software should consult the VMX capability MSR IA32_VMX_-MISC to determine the number supported (see Appendix A.6).

VM entry can be configured to conclude by delivering an event through the IDT (after all guest state and MSRs have been loaded). This process is called **event injection** and is controlled by the following three VM-entry control fields:

- **VM-entry interruption-information field** (32 bits). This field provides details about the event to be injected. Table 26-17 describes the field.

Table 26-17. Format of the VM-Entry Interruption-Information Field

Bit Position(s)	Content
7:0	Vector of interrupt or exception
10:8	Interruption type: 0: External interrupt 1: Reserved 2: Non-maskable interrupt (NMI) 3: Hardware exception (e.g,. #PF) 4: Software interrupt (INT *n*) 5: Privileged software exception (INT1) 6: Software exception (INT3 or INTO) 7: Other event
11	Deliver error code (0 = do not deliver; 1 = deliver)
30:12	Reserved
31	Valid

  — The **vector** (bits 7:0) determines which entry in the IDT is used or which other event is injected.

  — The **interruption type** (bits 10:8) determines details of how the injection is performed. In general, a VMM should use the type hardware exception for all exceptions **other than** the following:

    - breakpoint exceptions (#BP; a VMM should use the type software exception);
    - overflow exceptions (#OF a VMM should use the use type software exception); and
    - those debug exceptions (#DB) that are generated by INT1 (a VMM should use the use type privileged software exception).[1]

    The type **other event** is used for injection of events that are not delivered through the IDT.[2]

  — For exceptions, the **deliver-error-code bit** (bit 11) determines whether delivery pushes an error code on the guest stack.

  — VM entry injects an event if and only if the **valid bit** (bit 31) is 1. The valid bit in this field is cleared on every VM exit (see Section 29.2).

- **VM-entry exception error code** (32 bits). This field is used if and only if the valid bit (bit 31) and the deliver-error-code bit (bit 11) are both set in the VM-entry interruption-information field.
- **VM-entry instruction length** (32 bits). For injection of events whose type is software interrupt, software exception, or privileged software exception, this field is used to determine the value of RIP that is pushed on the stack.

See Section 28.6 for details regarding the mechanics of event injection, including the use of the interruption type and the VM-entry instruction length.

VM exits clear the valid bit (bit 31) in the VM-entry interruption-information field.

26.9 VM-EXIT INFORMATION FIELDS

The VMCS contains a section of fields that contain information about the most recent VM exit.

1. The type hardware exception should be used for all other debug exceptions.
2. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with values 1 or 3 for *n*.

On some processors, attempts to write to these fields with VMWRITE fail (see "VMWRITE—Write Field to Virtual-Machine Control Structure" in Chapter 31).[1]

26.9.1 Basic VM-Exit Information

The following VM-exit information fields provide basic information about a VM exit:

- **Exit reason** (32 bits). This field encodes the reason for the VM exit and has the structure given in Table 26-18.

Table 26-18. Format of Exit Reason

Bit Position(s)	Contents
15:0	Basic exit reason.
16	Always cleared to 0.
24:17	Not currently defined.
25	A VM exit saves this bit as 1 to indicate that the VM exit caused a shadow stack to become prematurely busy.
26	A VM exit saves this bit as 1 to indicate that the VM exit occurred after assertion of a bus lock while the "VMM bus-lock detection" VM-execution control was 1.
27	A VM exit saves this bit as 1 to indicate that the VM exit was incident to enclave mode.
28	Pending MTF VM exit.
29	VM exit from VMX root operation.
30	Not currently defined.
31	VM-entry failure (0 = true VM exit; 1 = VM-entry failure)

  - Bits 15:0 provide basic information about the cause of the VM exit (if bit 31 is clear) or of the VM-entry failure (if bit 31 is set). Appendix C enumerates the basic exit reasons.
  - Bit 16 is always cleared to 0.
  - Bit 25 is set to 1 if the "prematurely busy shadow stack" VM-exit control is 1 and the VM exit caused a shadow stack to become prematurely busy (see Section 27.4.3). Otherwise, the bit is cleared.
  - Bit 26 is set to 1 if the VM exit occurred after assertion of a bus lock while the "VMM bus-lock detection" VM-execution control was 1. Such VM exits include those that occur due to the 1-setting of that control as well as others that might occur during execution of an instruction that asserted a bus lock.
  - Bit 27 is set to 1 if the VM exit occurred while the logical processor was in enclave mode.

    A VM exit also sets this bit if it is incident to delivery of an event injected by VM entry and the guest interruptibility-state field indicates an enclave interrupt (bit 4 of the field is 1). See Section 29.2.1 for details.
  - Bit 28 is set only by an SMM VM exit (see Section 33.15.2) that took priority over an MTF VM exit (see Section 27.5.2) that would have occurred had the SMM VM exit not occurred. See Section 33.15.2.3.
  - Bit 29 is set if and only if the processor was in VMX root operation at the time the VM exit occurred. This can happen only for SMM VM exits. See Section 33.15.2.
  - Because some VM-entry failures load processor state from the host-state area (see Section 28.8), software must be able to distinguish such cases from true VM exits. Bit 31 is used for that purpose.
- **Exit qualification** (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field contains additional information about the cause of VM exits due to the following: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); task switches; INVEPT; INVLPG;INVVPID; LGDT; LIDT; LLDT; LTR; SGDT; SIDT; SLDT; STR; VMCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; XRSTORS; XSAVES; control-

1. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

register accesses; MOV DR; I/O instructions; and MWAIT. The format of the field depends on the cause of the VM exit. See Section 29.2.1 for details.

- **Guest-linear address** (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is used in the following cases:
  - VM exits due to attempts to execute LMSW with a memory operand.
  - VM exits due to attempts to execute INS or OUTS.
  - VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions.
  - Certain VM exits due to EPT violations

  See Section 29.2.1 and Section 33.15.2.3 for details of when and how this field is used.
- **Guest-physical address** (64 bits). This field is used by VM exits due to EPT violations and EPT misconfigurations. See Section 29.2.1 for details of when and how this field is used.

26.9.2 Information for VM Exits Due to Vectored Events

Event-specific information is provided for VM exits due to the following vectored events: exceptions (including those generated by the instructions INT3, INTO, INT1, BOUND, UD0, UD1, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs). This information is provided in the following fields:

- **VM-exit interruption information** (32 bits). This field receives basic information associated with the event causing the VM exit. Table 26-19 describes this field.

Table 26-19. Format of the VM-Exit Interruption-Information Field

Bit Position(s)	Content
7:0	Vector of interrupt or exception
10:8	Interruption type: 0: External interrupt 1: Not used 2: Non-maskable interrupt (NMI) 3: Hardware exception 4: Not used 5: Privileged software exception 6: Software exception 7: Not used
11	Error code valid (0 = invalid; 1 = valid)
12	NMI unblocking due to IRET
30:13	Not currently defined
31	Valid

- **VM-exit interruption error code** (32 bits). For VM exits caused by hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

Section 29.2.2 provides details of how these fields are saved on VM exits.

26.9.3 Information for VM Exits That Occur During Event Delivery

Additional information is provided for VM exits that occur during event delivery in VMX non-root operation.[1] This information is provided in the following fields:

1. This includes cases in which the event delivery was caused by event injection as part of VM entry; see Section 28.6.1.2.

- **IDT-vectoring information** (32 bits). This field receives basic information associated with the event that was being delivered when the VM exit occurred. Table 26-20 describes this field.

Table 26-20. Format of the IDT-Vectoring Information Field

Bit Position(s)	Content
7:0	Vector of interrupt or exception
10:8	Interruption type: 0: External interrupt 1: Not used 2: Non-maskable interrupt (NMI) 3: Hardware exception 4: Software interrupt 5: Privileged software exception 6: Software exception 7: Not used
11	Error code valid (0 = invalid; 1 = valid)
30:12	Not currently defined
31	Valid

- **IDT-vectoring error code** (32 bits). For VM exits the occur during delivery of hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

See Section 29.2.4 provides details of how these fields are saved on VM exits.

26.9.4 Information for VM Exits Due to Instruction Execution

The following fields are used for VM exits caused by attempts to execute certain instructions in VMX non-root operation:

- **VM-exit instruction length** (32 bits). For VM exits resulting from instruction execution, this field receives the length in bytes of the instruction whose execution led to the VM exit.[1] See Section 29.2.5 for details of when and how this field is used.
- **VM-exit instruction information** (32 bits). This field is used for VM exits due to attempts to execute INS, INVEPT, INVVPID, LIDT, LGDT, LLDT, LTR, OUTS, SIDT, SGDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, or VMXON.[2] The format of the field depends on the cause of the VM exit. See Section 29.2.5 for details.

The following fields (64 bits each; 32 bits on processors that do not support Intel 64 architecture) are used only for VM exits due to SMIs that arrive immediately after retirement of I/O instructions. They provide information about that I/O instruction:

- **I/O RCX**. The value of RCX before the I/O instruction started.
- **I/O RSI**. The value of RSI before the I/O instruction started.
- **I/O RDI**. The value of RDI before the I/O instruction started.
- **I/O RIP**. The value of RIP before the I/O instruction started (the RIP that addressed the I/O instruction).

An execution of WRMSRLIST causes a VM exit if it writes to an MSR that cannot be written due to the contents of the MSR bitmaps (see Section 27.1.3). Such VM exits save the data that would have been written to the MSR in a 64-bit field called **MSR data**. This field is supported only on processors that support the 1-setting of the “enable MSR-list instructions” VM-execution control.

1. This field is also used for VM exits that occur during the delivery of a software interrupt or software exception.
2. Whether the processor provides this information on VM exits due to attempts to execute INS or OUTS can be determined by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

26.9.5 VM-Instruction Error Field

The 32-bit **VM-instruction error field** does not provide information about the most recent VM exit. In fact, it is not modified on VM exits. Instead, it provides information about errors encountered by a non-faulting execution of one of the VMX instructions.

26.10 VMCS TYPES: ORDINARY AND SHADOW

Every VMCS is either an **ordinary VMCS** or a **shadow VMCS**. A VMCS's type is determined by the shadow-VMCS indicator in the VMCS region (this is the value of bit 31 of the first 4 bytes of the VMCS region; see Table 26-1): 0 indicates an ordinary VMCS, while 1 indicates a shadow VMCS. Shadow VMCSs are supported only on processors that support the 1-setting of the "VMCS shadowing" VM-execution control (see Section 26.6.2).

A shadow VMCS differs from an ordinary VMCS in two ways:

- An ordinary VMCS can be used for VM entry but a shadow VMCS cannot. Attempts to perform VM entry when the current VMCS is a shadow VMCS fail (see Section 28.1).
- The VMREAD and VMWRITE instructions can be used in VMX non-root operation to access a shadow VMCS but not an ordinary VMCS. This fact results from the following:
  - If the "VMCS shadowing" VM-execution control is 0, execution of the VMREAD and VMWRITE instructions in VMX non-root operation always cause VM exits (see Section 27.1.3).
  - If the "VMCS shadowing" VM-execution control is 1, execution of the VMREAD and VMWRITE instructions in VMX non-root operation can access the VMCS referenced by the VMCS link pointer (see Section 32.3).
  - If the "VMCS shadowing" VM-execution control is 1, VM entry ensures that any VMCS referenced by the VMCS link pointer is a shadow VMCS (see Section 28.3.1.5).

In VMX root operation, both types of VMCSs can be accessed with the VMREAD and VMWRITE instructions.

Software should not modify the shadow-VMCS indicator in the VMCS region of a VMCS that is active. Doing so may cause the VMCS to become corrupted (see Section 26.11.1). Before modifying the shadow-VMCS indicator, software should execute VMCLEAR for the VMCS to ensure that it is not active.

26.11 SOFTWARE USE OF THE VMCS AND RELATED STRUCTURES

This section details guidelines that software should observe when using a VMCS and related structures. It also provides descriptions of consequences for failing to follow guidelines.

26.11.1 Software Use of Virtual-Machine Control Structures

To ensure proper processor behavior, software should observe certain guidelines when using an active VMCS.

No VMCS should ever be active on more than one logical processor. If a VMCS is to be "migrated" from one logical processor to another, the first logical processor should execute VMCLEAR for the VMCS (to make it inactive on that logical processor and to ensure that all VMCS data are in memory) before the other logical processor executes VMPTRLD for the VMCS (to make it active on the second logical processor).[1] A VMCS that is made active on more than one logical processor may become **corrupted** (see below).

Software should not modify the shadow-VMCS indicator (see Table 26-1) in the VMCS region of a VMCS that is active. Doing so may cause the VMCS to become corrupted. Before modifying the shadow-VMCS indicator, software should execute VMCLEAR for the VMCS to ensure that it is not active.

Software should use the VMREAD and VMWRITE instructions to access the different fields in the current VMCS (see Section 26.11.2). Software should never access or modify the VMCS data of an active VMCS using ordinary

1. As noted in Section 26.1, execution of the VMPTRLD instruction makes a VMCS is active. In addition, VM entry makes active any shadow VMCS referenced by the VMCS link pointer in the current VMCS. If a shadow VMCS is made active by VM entry, it is necessary to execute VMCLEAR for that VMCS before allowing that VMCS to become active on another logical processor.

memory operations, in part because the format used to store the VMCS data is implementation-specific and not architecturally defined, and also because a logical processor may maintain some VMCS data of an active VMCS on the processor and not in the VMCS region. The following items detail some of the hazards of accessing VMCS data using ordinary memory operations:

- Any data read from a VMCS with an ordinary memory read does not reliably reflect the state of the VMCS. Results may vary from time to time or from logical processor to logical processor.
- Writing to a VMCS with an ordinary memory write is not guaranteed to have a deterministic effect on the VMCS. Doing so may cause the VMCS to become corrupted (see below).

(Software can avoid these hazards by removing any linear-address mappings to a VMCS region before executing a VMPTRLD for that region and by not remapping it until after executing VMCLEAR for that region.)

If a logical processor leaves VMX operation, any VMCSs active on that logical processor may be corrupted (see below). To prevent such corruption of a VMCS that may be used either after a return to VMX operation or on another logical processor, software should execute VMCLEAR for that VMCS before executing the VMXOFF instruction or removing power from the processor (e.g., as part of a transition to the S3 and S4 power states).

This section has identified operations that may cause a VMCS to become corrupted. These operations may cause the VMCS's data to become undefined. Behavior may be unpredictable if that VMCS used subsequently on any logical processor. The following items detail some hazards of VMCS corruption:

- VM entries may fail for unexplained reasons or may load undesired processor state.
- The processor may not correctly support VMX non-root operation as documented in Chapter 26 and may generate unexpected VM exits.
- VM exits may load undesired processor state, save incorrect state into the VMCS, or cause the logical processor to transition to a shutdown state.

26.11.2 VMREAD, VMWRITE, and Encodings of VMCS Fields

Every field of the VMCS is associated with a 32-bit value that is its **encoding**. The encoding is provided in an operand to VMREAD and VMWRITE when software wishes to read or write that field. These instructions fail if given, in 64-bit mode, an operand that sets an encoding bit beyond bit 32. See Chapter 31 for a description of these instructions.

The structure of the 32-bit encodings of the VMCS components is determined principally by the width of the fields and their function in the VMCS. See Table 26-21.

Table 26-21. Structure of VMCS Component Encoding

Bit Position(s)	Contents
0	Access type (0 = full; 1 = high); must be full for 16-bit, 32-bit, and natural-width fields
9:1	Index
11:10	Type: 0: control 1: VM-exit information 2: guest state 3: host state
12	Reserved (must be 0)
14:13	Width: 0: 16-bit 1: 64-bit 2: 32-bit 3: natural-width
31:15	Reserved (must be 0)

The following items detail the meaning of the bits in each encoding:

- **Field width.** Bits 14:13 encode the width of the field.
  - A value of 0 indicates a 16-bit field.
  - A value of 1 indicates a 64-bit field.
  - A value of 2 indicates a 32-bit field.
  - A value of 3 indicates a **natural-width** field. Such fields have 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not.

  Fields whose encodings use value 1 are specially treated to allow 32-bit software access to all 64 bits of the field. Such access is allowed by defining, for each such field, an encoding that allows direct access to the high 32 bits of the field. See below.
- **Field type.** Bits 11:10 encode the type of VMCS field: control, guest-state, host-state, or VM-exit information. (The last category also includes the VM-instruction error field.)
- **Index.** Bits 9:1 distinguish components with the same field width and type.
- **Access type.** Bit 0 must be 0 for all fields except for 64-bit fields (those with field-width 1; see above). A VMREAD or VMWRITE using an encoding with this bit cleared to 0 accesses the entire field. For a 64-bit field with field-width 1, a VMREAD or VMWRITE using an encoding with this bit set to 1 accesses only the high 32 bits of the field.

Appendix B gives the encodings of all fields in the VMCS.

The following describes the operation of VMREAD and VMWRITE based on processor mode, VMCS-field width, and access type:

- 16-bit fields:
  - A VMREAD returns the value of the field in bits 15:0 of the destination operand; other bits of the destination operand are cleared to 0.
  - A VMWRITE writes the value of bits 15:0 of the source operand into the VMCS field; other bits of the source operand are not used.
- 32-bit fields:
  - A VMREAD returns the value of the field in bits 31:0 of the destination operand; in 64-bit mode, bits 63:32 of the destination operand are cleared to 0.
  - A VMWRITE writes the value of bits 31:0 of the source operand into the VMCS field; in 64-bit mode, bits 63:32 of the source operand are not used.
- 64-bit fields and natural-width fields using the full access type outside IA-32e mode.
  - A VMREAD returns the value of bits 31:0 of the field in its destination operand; bits 63:32 of the field are ignored.
  - A VMWRITE writes the value of its source operand to bits 31:0 of the field and clears bits 63:32 of the field.
- 64-bit fields and natural-width fields using the full access type in 64-bit mode (only on processors that support Intel 64 architecture).
  - A VMREAD returns the value of the field in bits 63:0 of the destination operand
  - A VMWRITE writes the value of bits 63:0 of the source operand into the VMCS field.
- 64-bit fields using the high access type.
  - A VMREAD returns the value of bits 63:32 of the field in bits 31:0 of the destination operand; in 64-bit mode, bits 63:32 of the destination operand are cleared to 0.
  - A VMWRITE writes the value of bits 31:0 of the source operand to bits 63:32 of the field; in 64-bit mode, bits 63:32 of the source operand are not used.

Software seeking to read a 64-bit field outside IA-32e mode can use VMREAD with the full access type (reading bits 31:0 of the field) and VMREAD with the high access type (reading bits 63:32 of the field); the order of the two VMREAD executions is not important. Software seeking to modify a 64-bit field outside IA-32e mode should first use VMWRITE with the full access type (establishing bits 31:0 of the field while clearing bits 63:32) and then use VMWRITE with the high access type (establishing bits 63:32 of the field).

26.11.3 Initializing a VMCS

Software should initialize fields in a VMCS (using VMWRITE) before using the VMCS for VM entry. Failure to do so may result in unpredictable behavior; for example, a VM entry may fail for unexplained reasons, or a successful transition (VM entry or VM exit) may load processor state with unexpected values.

It is not necessary to initialize fields that the logical processor will not use. (For example, it is not necessary to initialize the MSR-bitmap address if the "use MSR bitmaps" VM-execution control is 0.)

A processor maintains some VMCS information that cannot be modified with the VMWRITE instruction; this includes a VMCS's launch state (see Section 26.1). Such information may be stored in the VMCS data portion of a VMCS region. Because the format of this information is implementation-specific, there is no way for software to know, when it first allocates a region of memory for use as a VMCS region, how the processor will determine this information from the contents of the memory region.

In addition to its other functions, the VMCLEAR instruction initializes any implementation-specific information in the VMCS region referenced by its operand. To avoid the uncertainties of implementation-specific behavior, software should execute VMCLEAR on a VMCS region before making the corresponding VMCS active with VMPTRLD for the first time. (Figure 26-1 illustrates how execution of VMCLEAR puts a VMCS into a well-defined state.)

The following software usage is consistent with these limitations:

- VMCLEAR should be executed for a VMCS before it is used for VM entry for the first time.
- VMLAUNCH should be used for the first VM entry using a VMCS after VMCLEAR has been executed for that VMCS.
- VMRESUME should be used for any subsequent VM entry using a VMCS (until the next execution of VMCLEAR for the VMCS).

It is expected that, in general, VMRESUME will have lower latency than VMLAUNCH. Since "migrating" a VMCS from one logical processor to another requires use of VMCLEAR (see Section 26.11.1), which sets the launch state of the VMCS to "clear", such migration requires the next VM entry to be performed using VMLAUNCH. Software developers can avoid the performance cost of increased VM-entry latency by avoiding unnecessary migration of a VMCS from one logical processor to another.

26.11.4 Software Access to Related Structures

In addition to data in the VMCS region itself, VMX non-root operation can be controlled by data structures that are referenced by pointers in a VMCS (for example, the I/O bitmaps). While the pointers to these data structures are parts of the VMCS, the data structures themselves are not. They are not accessible using VMREAD and VMWRITE but by ordinary memory writes.

Software should ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 26.11.1). Exceptions are made for the following data structures (subject to detailed discussion in the sections indicated): EPT paging structures and the data structures used to locate SPP vectors (Section 30.4.3); the virtual-APIC page (Section 31.1); the posted interrupt descriptor (Section 31.6); and the virtualization-exception information area (Section 27.5.7.2).

26.11.5 VMXON Region

Before executing VMXON, software allocates a region of memory (called the VMXON region)[1] that the logical processor uses to support VMX operation. The physical address of this region (the VMXON pointer) is provided in an operand to VMXON. The VMXON pointer is subject to the limitations that apply to VMCS pointers:

1. The amount of memory required for the VMXON region is the same as that required for a VMCS region. This size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

- The VMXON pointer must be 4-KByte aligned (bits 11:0 must be zero).
- The VMXON pointer must not set any bits beyond the processor's physical-address width.[1,2]

Before executing VMXON, software should write the VMCS revision identifier (see Section 26.2) to the VMXON region. (Specifically, it should write the 31-bit VMCS revision identifier to bits 30:0 of the first 4 bytes of the VMXON region; bit 31 should be cleared to 0.) It need not initialize the VMXON region in any other way. Software should use a separate region for each logical processor and should not access or modify the VMXON region of a logical processor between execution of VMXON and VMXOFF on that logical processor. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 26.11.1).

1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
2. If IA32_VMX_BASIC[48] is read as 1, the VMXON pointer must not set any bits in the range 63:32; see Appendix A.1.

13. Updates to Chapter 27, Volume 3C

Change bars and violet text show changes to Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C:* System Programming Guide, Part 3.

Changes to this chapter:

- Added information for the MSR data field of VMCS to WRMSRLIST in Section 27.1.3, “Instructions That Cause VM Exits Conditionally.”

CHAPTER 27
VMX NON-ROOT OPERATION

In a virtualized environment using VMX, the guest software stack typically runs on a logical processor in VMX non-root operation. This mode of operation is similar to that of ordinary processor operation outside of the virtualized environment. This chapter describes the differences between VMX non-root operation and ordinary processor operation with special attention to causes of VM exits (which bring a logical processor from VMX non-root operation to root operation). The differences between VMX non-root operation and ordinary processor operation are described in the following sections:

- Section 27.1, "Instructions That Cause VM Exits."
- Section 27.2, "Other Causes of VM Exits."
- Section 27.3, "Changes to Instruction Behavior in VMX Non-Root Operation."
- Section 27.4, "Other Changes in VMX Non-Root Operation."
- Section 27.5, "Features Specific to VMX Non-Root Operation."
- Section 27.6, "Unrestricted Guests."

Chapter 27, "VMX Non-Root Operation," describes the data control structures that govern VMX non-root operation. Chapter 27, "VMX Non-Root Operation," describes the operation of VM entries by which the processor transitions from VMX root operation to VMX non-root operation. Chapter 27, "VMX Non-Root Operation," describes the operation of VM exits by which the processor transitions from VMX non-root operation to VMX root operation.

Chapter 30, "VMX Support for Address Translation," describes two features that support address translation in VMX non-root operation. Chapter 31, "APIC Virtualization and Virtual Interrupts," describes features that support virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC) in VMX non-root operation.

27.1 INSTRUCTIONS THAT CAUSE VM EXITS

Certain instructions may cause VM exits if executed in VMX non-root operation. Unless otherwise specified, such VM exits are "fault-like," meaning that the instruction causing the VM exit does not execute and no processor state is updated by the instruction. Section 29.1 details architectural state in the context of a VM exit.

Section 27.1.1 defines the prioritization between faults and VM exits for instructions subject to both. Section 27.1.2 identifies instructions that cause VM exits whenever they are executed in VMX non-root operation (and thus can never be executed in VMX non-root operation). Section 27.1.3 identifies instructions that cause VM exits depending on the settings of certain VM-execution control fields (see Section 26.6).

27.1.1 Relative Priority of Faults and VM Exits

The following principles describe the ordering between existing faults and VM exits:

- Certain exceptions have priority over VM exits. These include invalid-opcode exceptions, faults based on privilege level,[1] and general-protection exceptions that are based on checking I/O permission bits in the task-state segment (TSS). For example, execution of RDMSR with CPL = 3 generates a general-protection exception and not a VM exit.[2]
- Faults incurred while fetching instruction operands have priority over VM exits that are conditioned based on the contents of those operands (see LMSW in Section 27.1.3).
- VM exits caused by execution of the INS and OUTS instructions (resulting either because the "unconditional I/O exiting" VM-execution control is 1 or because the "use I/O bitmaps control is 1) have priority over the following faults:

1. These include faults generated by attempts to execute, in virtual-8086 mode, privileged instructions that are not recognized in that mode.
2. MOV DR is an exception to this rule; see Section 27.1.3.

- A general-protection fault due to the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) being unusable
- A general-protection fault due to an offset beyond the limit of the relevant segment
- An alignment-check exception

• Fault-like VM exits have priority over exceptions other than those mentioned above. For example, RDMSR of a non-existent MSR with CPL = 0 generates a VM exit and not a general-protection exception.

When Section 27.1.2 or Section 27.1.3 (below) identify an instruction execution that may lead to a VM exit, it is assumed that the instruction does not incur a fault that takes priority over a VM exit.

27.1.2 Instructions That Cause VM Exits Unconditionally

The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, GETSEC,[1] INVD, and XSETBV. This is also true of instructions introduced with VMX: INVEPT, INVVPID, VMCALL,[2] VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, and VMXON.

27.1.3 Instructions That Cause VM Exits Conditionally

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause "fault-like" VM exits based on the conditions described:[3]

- **CLTS.** The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.
- **ENCLS.** The ENCLS instruction causes a VM exit if the "enable ENCLS exiting" VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the ENCLS-exiting bitmap is 1 (see Section 26.6.16).
  - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLS-exiting bitmap is 1.
- **ENCLV.** The ENCLV instruction causes a VM exit if the "enable ENCLV exiting" VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the ENCLV-exiting bitmap is 1 (see Section 26.6.17).
  - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLV-exiting bitmap is 1.
- **ENQCMD, ENQCMDS.** The behavior of each of these instructions is determined by the setting of the "PASID translation" VM-execution control. If that control is 0, the instruction executes normally. If the control is 1, instruction behavior is modified and may cause a VM exit. See Section 27.5.8.
- **HLT.** The HLT instruction causes a VM exit if the "HLT exiting" VM-execution control is 1.
- **IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD.** The behavior of each of these instructions is determined by the settings of the "unconditional I/O exiting" and "use I/O bitmaps" VM-execution controls:
  - If both controls are 0, the instruction executes normally.

1. An execution of GETSEC in VMX non-root operation causes a VM exit if CR4.SMXE[Bit 14] = 1 regardless of the value of CPL or RAX. An execution of GETSEC causes an invalid-opcode exception (#UD) if CR4.SMXE[Bit 14] = 0.
2. Under the dual-monitor treatment of SMIs and SMM, executions of VMCALL cause SMM VM exits in VMX root operation outside SMM. See Section 33.15.2.
3. Items in this section may refer to secondary processor-based VM-execution controls and tertiary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the secondary processor-based VM-execution controls were all 0; similarly, if bit 17 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the tertiary processor-based VM-execution controls were all 0. See Section 26.6.2.

- If the "unconditional I/O exiting" VM-execution control is 1 and the "use I/O bitmaps" VM-execution control is 0, the instruction causes a VM exit.
- If the "use I/O bitmaps" VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap (see Section 26.6.4). If an I/O operation "wraps around" the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the "unconditional I/O exiting" VM-execution control is ignored if the "use I/O bitmaps" VM-execution control is 1).

See Section 27.1.1 for information regarding the priority of VM exits relative to faults that may be caused by the INS and OUTS instructions.

- **INVLPG.** The INVLPG instruction causes a VM exit if the "INVLPG exiting" VM-execution control is 1.
- **INVPCID.** The INVPCID instruction causes a VM exit if the "INVLPG exiting" and "enable INVPCID" VM-execution controls are both 1.
- **LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR.** These instructions cause VM exits if the "descriptor-table exiting" VM-execution control is 1.
- **LMSW.** In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. LMSW never clears bit 0 of CR0 (CR0.PE); thus, LMSW causes a VM exit if either of the following are true:
  - The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/host mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.
  - For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/host mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.
- **LOADIWKEY.** The LOADIWKEY instruction causes a VM exit if the "LOADIWKEY exiting" VM-execution control is 1.
- **MONITOR.** The MONITOR instruction causes a VM exit if the "MONITOR exiting" VM-execution control is 1.
- **MOV from CR3.** The MOV from CR3 instruction causes a VM exit if the "CR3-store exiting" VM-execution control is 1. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
- **MOV from CR8.** The MOV from CR8 instruction causes a VM exit if the "CR8-store exiting" VM-execution control is 1.
- **MOV to CR0.** The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow. (If every bit is clear in the CR0 guest/host mask, MOV to CR0 cannot cause a VM exit.)
- **MOV to CR3.** The MOV to CR3 instruction causes a VM exit unless the "CR3-load exiting" VM-execution control is 0 or the value of its source operand is equal to one of the CR3-target values specified in the VMCS. Only the first *n* CR3-target values are considered, where *n* is the CR3-target count. If the "CR3-load exiting" VM-execution control is 1 and the CR3-target count is 0, MOV to CR3 always causes a VM exit.

  The first processors to support the virtual-machine extensions supported only the 1-setting of the "CR3-load exiting" VM-execution control. These processors always consult the CR3-target controls to determine whether an execution of MOV to CR3 causes a VM exit.
- **MOV to CR4.** The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.
- **MOV to CR8.** The MOV to CR8 instruction causes a VM exit if the "CR8-load exiting" VM-execution control is 1.
- **MOV DR.** The MOV DR instruction causes a VM exit if the "MOV-DR exiting" VM-execution control is 1. Such VM exits represent an exception to the principles identified in Section 27.1.1 in that they take priority over the following: general-protection exceptions based on privilege level; and invalid-opcode exceptions that occur because CR4.DE=1 and the instruction specified access to DR4 or DR5.
- **MWAIT.** The MWAIT instruction causes a VM exit if the "MWAIT exiting" VM-execution control is 1. If this control is 0, the behavior of the MWAIT instruction may be modified (see Section 27.3).
- **PAUSE.** The behavior of each of this instruction depends on CPL and the settings of the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls**:**
  - CPL = 0.

    - If the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls are both 0, the PAUSE instruction executes normally.
    - If the "PAUSE exiting" VM-execution control is 1, the PAUSE instruction causes a VM exit (the "PAUSE-loop exiting" VM-execution control is ignored if CPL = 0 and the "PAUSE exiting" VM-execution control is 1).
    - If the "PAUSE exiting" VM-execution control is 0 and the "PAUSE-loop exiting" VM-execution control is 1, the following treatment applies.

      The processor determines the amount of time between this execution of PAUSE and the previous execution of PAUSE at CPL 0. If this amount of time exceeds the value of the VM-execution control field PLE_Gap, the processor considers this execution to be the first execution of PAUSE in a loop. (It also does so for the first execution of PAUSE at CPL 0 after VM entry.)

      Otherwise, the processor determines the amount of time since the most recent execution of PAUSE that was considered to be the first in a loop. If this amount of time exceeds the value of the VM-execution control field PLE_Window, a VM exit occurs.

      For purposes of these computations, time is measured based on a counter that runs at the same rate as the timestamp counter (TSC).
  - CPL > 0.
    - If the "PAUSE exiting" VM-execution control is 0, the PAUSE instruction executes normally.
    - If the "PAUSE exiting" VM-execution control is 1, the PAUSE instruction causes a VM exit.

    The "PAUSE-loop exiting" VM-execution control is ignored if CPL > 0.
- **PCONFIG.** The PCONFIG instruction causes a VM exit if the "enable PCONFIG" VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the PCONFIG-exiting bitmap is 1 (see Section 26.6.18).
  - The value of EAX is greater than or equal to 63 and bit 63 in the PCONFIG-exiting bitmap is 1.

  If the "enable PCONFIG" VM-execution control is 1 and neither of the previous items hold, the PCONFIG instruction executes normally.
- **RDMSR.** The RDMSR instruction causes a VM exit if any of the following are true:
  - The "use MSR bitmaps" VM-execution control is 0.
  - The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  - The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in read bitmap for low MSRs is 1, where *n* is the value of ECX.
  - The value of ECX is in the range C0000000H – C0001FFFH and bit *n* in read bitmap for high MSRs is 1, where *n* is the value of ECX & 00001FFFH.

  See Section 26.6.9 for details regarding how these bitmaps are identified.
- **RDMSRLIST.** The RDMSRLIST instruction causes a VM exit if the "enable MSR-list instructions" VM-execution control is 1 and the "use MSR bitmaps" VM-execution control is 0. If both controls are 1, the instruction reads one MSR at a time normally, storing the value read to memory and clearing the corresponding bit in RCX. An attempt to read MSR X causes a VM exit if any of the following are true:
  - X is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  - X is in the range 00000000H – 00001FFFH and bit X in read bitmap for low MSRs is 1.
  - X is in the range C0000000H – C0001FFFH and bit *n* in read bitmap for high MSRs is 1, where *n* is the value of X & 00001FFFH.

  If an attempt to read an MSR causes a VM exit, the corresponding bit in RCX is not cleared, the MSR is not read, and no value is stored to memory.
- **RDPMC.** The RDPMC instruction causes a VM exit if the "RDPMC exiting" VM-execution control is 1.
- **RDRAND.** The RDRAND instruction causes a VM exit if the "RDRAND exiting" VM-execution control is 1.

- **RDSEED.** The RDSEED instruction causes a VM exit if the "RDSEED exiting" VM-execution control is 1.
- **RDTSC.** The RDTSC instruction causes a VM exit if the "RDTSC exiting" VM-execution control is 1.
- **RDTSCP.** The RDTSCP instruction causes a VM exit if the "RDTSC exiting" and "enable RDTSCP" VM-execution controls are both 1.
- **RSM.** The RSM instruction causes a VM exit if executed in system-management mode (SMM).[1]
- **TPAUSE.** The TPAUSE instruction causes a VM exit if the "RDTSC exiting" and "enable user wait and pause" VM-execution controls are both 1.
- **UMWAIT.** The UMWAIT instruction causes a VM exit if the "RDTSC exiting" and "enable user wait and pause" VM-execution controls are both 1.
- **VMREAD.** The VMREAD instruction causes a VM exit if any of the following are true:
  - The "VMCS shadowing" VM-execution control is 0.
  - Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  - Bit *n* in VMREAD bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 26.6.15 for details regarding how the VMREAD bitmap is identified.

  If the VMREAD instruction does not cause a VM exit, it reads from the VMCS referenced by the VMCS link pointer. See Chapter 32, "VMREAD—Read Field from Virtual-Machine Control Structure" for details of the operation of the VMREAD instruction.
- **VMWRITE.** The VMWRITE instruction causes a VM exit if any of the following are true:
  - The "VMCS shadowing" VM-execution control is 0.
  - Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  - Bit *n* in VMWRITE bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 26.6.15 for details regarding how the VMWRITE bitmap is identified.

  If the VMWRITE instruction does not cause a VM exit, it writes to the VMCS referenced by the VMCS link pointer. See Chapter 32, "VMWRITE—Write Field to Virtual-Machine Control Structure" for details of the operation of the VMWRITE instruction.
- **WBINVD.** The WBINVD instruction causes a VM exit if the "WBINVD exiting" VM-execution control is 1.
- **WBNOINVD.** The WBNOINVD instruction causes a VM exit if the "WBINVD exiting" VM-execution control is 1.
- **WRMSR, WRMSRNS.** Execution of one of these instructions causes a VM exit if any of the following are true:
  - The "use MSR bitmaps" VM-execution control is 0.
  - The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  - The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in write bitmap for low MSRs is 1, where *n* is the value of ECX.
  - The value of ECX is in the range C0000000H – C0001FFFH and bit *n* in write bitmap for high MSRs is 1, where *n* is the value of ECX & 00001FFFH.

  See Section 26.6.9 for details regarding how these bitmaps are identified.
- **WRMSRLIST.** The WRMSRLIST instruction causes a VM exit if the "enable MSR-list instructions" VM-execution control is 1 and the "use MSR bitmaps" VM-execution control is 0. In this case, the register operands of the instructions are not used, memory is not read, and no MSRs are modified.

  If both controls are 1, the instruction writes one MSR at a time normally, using a value read from memory and clearing the corresponding bit in RCX. An attempt to write MSR X causes a VM exit if any of the following are true:
  - X is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  - X is in the range 00000000H – 00001FFFH and bit X in write bitmap for low MSRs is 1.

1. Execution of the RSM instruction outside SMM causes an invalid-opcode exception regardless of whether the processor is in VMX operation. It also does so in VMX root operation in SMM; see Section 33.15.3.

— X is in the range C0000000H – C0001FFFH and bit *n* in write bitmap for high MSRs is 1, where *n* is the value of X & 00001FFFH.

Such a VM exit occurs after the data that would be written to the MSR is read from memory, but the corresponding bit in RCX is not cleared and the MSR is not written. The data that would have been written to the MSR is saved into the MSR-data field of the VMCS (see Section 29.2.5).

- **XRSTORS.** The XRSTORS instruction causes a VM exit if the "enable XSAVES/XRSTORS" VM-execution control is 1and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 26.6.21).
- **XSAVES.** The XSAVES instruction causes a VM exit if the "enable XSAVES/XRSTORS" VM-execution control is 1 and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 26.6.21).

27.2 OTHER CAUSES OF VM EXITS

In addition to VM exits caused by instruction execution, the following events can cause VM exits:[1]

- **Exceptions.** Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap (see Section 26.6.3). If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT1, INT3, INTO, BOUND, UD0, UD1, and UD2.[2]

  Page faults (exceptions with vector 14) are specially treated. When a page fault occurs, a processor consults (1) bit 14 of the exception bitmap; (2) the error code produced with the page fault [PFEC]; (3) the page-fault error-code mask field [PFEC_MASK]; and (4) the page-fault error-code match field [PFEC_MATCH]. It checks if PFEC & PFEC_MASK = PFEC_MATCH. If there is equality, the specification of bit 14 in the exception bitmap is followed (for example, a VM exit occurs if that bit is set). If there is inequality, the meaning of that bit is reversed (for example, a VM exit occurs if that bit is clear).

  Thus, if software desires VM exits on all page faults, it can set bit 14 in the exception bitmap to 1 and set the page-fault error-code mask and match fields each to 00000000H. If software desires VM exits on no page faults, it can set bit 14 in the exception bitmap to 1, the page-fault error-code mask field to 00000000H, and the page-fault error-code match field to FFFFFFFFH.
- **Triple fault.** A VM exit occurs if the logical processor encounters an exception while attempting to call the double-fault handler and that exception itself does not cause a VM exit due to the exception bitmap. This applies to the case in which the double-fault exception was generated within VMX non-root operation, the case in which the double-fault exception was generated during event injection by VM entry, and to the case in which VM entry is injecting a double-fault exception.
- **External interrupts.** An external interrupt causes a VM exit if the "external-interrupt exiting" VM-execution control is 1 (see Section 31.6 for an exception.) Otherwise, the processor handles the interrupt is normally.[3] (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The processor does handle the interrupt and no VM exit occurs.)
- **Non-maskable interrupts (NMIs).** An NMI causes a VM exit if the "NMI exiting" VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)
- **INIT signals.** INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would

1. Items in this section may refer to secondary processor-based VM-execution controls and tertiary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the secondary processor-based VM-execution controls were all 0; similarly, if bit 17 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the tertiary processor-based VM-execution controls were all 0. See Section 26.6.2.
2. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with value 1 or 3 for n.
3. Normal handling usually means delivery through the IDT, but it could also mean treatment of the interrupt as a user-interrupt notification.

outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)

- **Start-up IPIs (SIPIs). SIPIs cause VM exits.** If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for-SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)
- **Task switches.** Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. See Section 27.4.2.
- **System-management interrupts (SMIs).** If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits. See Section 33.15.2.[1]
- **VMX-preemption timer.** A VM exit occurs when the timer counts down to zero. See Section 27.5.1 for details of operation of the VMX-preemption timer.

  Debug-trap exceptions and higher priority events take priority over VM exits caused by the VMX-preemption timer. VM exits caused by the VMX-preemption timer take priority over VM exits caused by the "NMI-window exiting" VM-execution control and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would a non-maskable interrupt. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.
- **Bus locks.** Assertion of a bus lock (see Section 10.1.2) causes a VM exit if the "VMM bus-lock detection" VM-execution control is 1. Such a VM exit is trap-like because it is generated after execution of an instruction that asserts a bus lock. The VM exit thus does not prevent assertion of the bus lock. These VM exits take priority over system-management interrupts (SMIs), INIT signals, and lower priority events.
- **Instruction timeout.** If the "instruction timeout" VM-execution control is 1, a VM exit occurs if certain operations prevent the processor from reaching an instruction boundary within the amount of time specified by the instruction-timeout control VM-execution control field (see Section 26.6.25).

In addition, there are controls that cause VM exits based on the readiness of guest software to receive interrupts:

- If the "interrupt-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if RFLAGS.IF = 1 and there is no blocking of events by STI or by MOV SS (see Table 26-3).

  Non-maskable interrupts (NMIs) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over external interrupts and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would an external interrupt. Specifically, they wake a logical processor from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the shutdown state or the wait-for-SIPI state.
- If the "NMI-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if there is no virtual-NMI blocking and there is no blocking of events by MOV SS and no blocking of events by STI (see Table 26-3).

  VM exits caused by the VMX-preemption timer and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non-maskable interrupts (NMIs) and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would an NMI. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

Conditions necessary for some of the VM exits identified in this section may hold immediately after VM entry. If they do, a corresponding VM exit occurs at that time.

1. Under the dual-monitor treatment of SMIs and SMM, SMIs also cause SMM VM exits if they occur in VMX root operation outside SMM. If the processor is using the default treatment of SMIs and SMM, SMIs are delivered as described in Section 33.14.1.

27.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:[1]

- **CLTS.** Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  - If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation (see Section 25.8), in which case CLTS causes a general-protection exception.
  - If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.
  - If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit.
- **ENQCMD, ENQCMDS.** Each of these instructions performs a 64-byte enqueue store that includes a PASID value in bits 19:0. For ENQCMD, the PASID is normally the value of IA32_PASID[19:0], while for ENQCMDS, the PASID is normally read from memory.

  The behavior of each of these instructions (and in particular the PASID value used for the enqueue store) is determined by the setting of the "PASID translation" VM-execution control:
  - If the "PASID translation" VM-execution control is 0, the instruction operates normally.
  - If the "PASID translation" VM-execution control is 1, the PASID value used for the enqueue store is determined by the PASID-translation process described in Section 27.5.8. (Note the PASID translation may result in a VM exit, in which case the enqueue store is not performed.)

  An execution of ENQCMD or ENQCMDS performs PASID translation only after checking for conditions that may result in general-protection exception (the check of IA32_PASID.Valid for ENQCMD; the privilege-level check for ENQCMDS), after loading the instruction's source operand from memory, and thus after any faults or VM exits that the loading may cause (e.g., page faults or EPT violations). PASID translation occurs before the actual enqueue store and thus before any faults or VM exits that it may cause.
- **INVPCID.** Behavior of the INVPCID instruction is determined first by the setting of the "enable INVPCID" VM-execution control:
  - If the "enable INVPCID" VM-execution control is 0, INVPCID causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.
  - If the "enable INVPCID" VM-execution control is 1, treatment is based on the setting of the "INVLPG exiting" VM-execution control:
    - If the "INVLPG exiting" VM-execution control is 0, INVPCID operates normally.
    - If the "INVLPG exiting" VM-execution control is 1, INVPCID causes a VM exit.
- **IRET.** Behavior of IRET with regard to NMI blocking (see Table 26-3) is determined by the settings of the "NMI exiting" and "virtual NMIs" VM-execution controls:
  - If the "NMI exiting" VM-execution control is 0, IRET operates normally and unblocks NMIs. (If the "NMI exiting" VM-execution control is 0, the "virtual NMIs" control must be 0; see Section 28.2.1.1.)
  - If the "NMI exiting" VM-execution control is 1, IRET does not affect blocking of NMIs. If, in addition, the "virtual NMIs" VM-execution control is 1, the logical processor tracks virtual-NMI blocking. In this case, IRET removes any virtual-NMI blocking.

  The unblocking of NMIs or virtual NMIs specified above occurs even if IRET causes a fault.
- **LMSW.** Outside of VMX non-root operation, LMSW loads its source operand into CR0[3:0], but it does not clear CR0.PE if that bit is set. In VMX non-root operation, an execution of LMSW that does not cause a VM exit (see Section 27.1.3) leaves unmodified any bit in CR0[3:0] corresponding to a bit set in the CR0 guest/host mask.

1. Items in this section may refer to secondary processor-based VM-execution controls and tertiary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the secondary processor-based VM-execution controls were all 0; similarly, if bit 17 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the tertiary processor-based VM-execution controls were all 0. See Section 26.6.2.

An attempt to set any other bit in CR0[3:0] to a value not supported in VMX operation (see Section 25.8) causes a general-protection exception. Attempts to clear CR0.PE are ignored without fault.

- **MOV from CR0.** The behavior of MOV from CR0 is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow.

  Depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.
- **MOV from CR3.** If the "enable EPT" VM-execution control is 1 and an execution of MOV from CR3 does not cause a VM exit (see Section 27.1.3), the value loaded from CR3 is a guest-physical address; see Section 30.3.1.
- **MOV from CR4.** The behavior of MOV from CR4 is determined by the CR4 guest/host mask and the CR4 read shadow. For each position corresponding to a bit clear in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR4. For each position corresponding to a bit set in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR4 read shadow. Thus, if every bit is cleared in the CR4 guest/host mask, MOV from CR4 reads normally from CR4; if every bit is set in the CR4 guest/host mask, MOV from CR4 returns the value of the CR4 read shadow.

  Depending on the contents of the CR4 guest/host mask and the CR4 read shadow, bits may be set in the destination that would never be set when reading directly from CR4.
- **MOV from CR8.** If the MOV from CR8 instruction does not cause a VM exit (see Section 27.1.3), its behavior is modified if the "use TPR shadow" VM-execution control is 1; see Section 31.3.
- **MOV to CR0.** An execution of MOV to CR0 that does not cause a VM exit (see Section 27.1.3) leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. Treatment of attempts to modify other bits in CR0 depends on the setting of the "unrestricted guest" VM-execution control:
  - If the control is 0, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 to a value not supported in VMX operation (see Section 25.8).
  - If the control is 1, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 other than bit 0 (PE) or bit 31 (PG) to a value not supported in VMX operation. It remains the case, however, that MOV to CR0 causes a general-protection exception if it would result in CR0.PE = 0 and CR0.PG = 1 or if it would result in CR0.PG = 1, CR4.PAE = 0, and IA32_EFER.LME = 1.
- **MOV to CR3.** If the "enable EPT" VM-execution control is 1 and an execution of MOV to CR3 does not cause a VM exit (see Section 27.1.3), the value loaded into CR3 is treated as a guest-physical address; see Section 30.3.1.
  - If PAE paging is not being used, the instruction does not use the guest-physical address to access memory and it does not cause it to be translated through EPT.[1]
  - If PAE paging is being used, the instruction translates the guest-physical address through EPT and uses the result to load the four (4) page-directory-pointer-table entries (PDPTEs). The instruction does not use the guest-physical addresses the PDPTEs to access memory and it does not cause them to be translated through EPT.
- **MOV to CR4.** An execution of MOV to CR4 that does not cause a VM exit (see Section 27.1.3) leaves unmodified any bit in CR4 corresponding to a bit set in the CR4 guest/host mask. Such an execution causes a general-protection exception if it attempts to set any bit in CR4 (not corresponding to a bit set in the CR4 guest/host mask) to a value not supported in VMX operation (see Section 25.8).
- **MOV to CR8.** If the MOV **to** CR8 instruction does not cause a VM exit (see Section 27.1.3), its behavior is modified if the "use TPR shadow" VM-execution control is 1; see Section 31.3.
- **MWAIT.** Behavior of the MWAIT instruction (which always causes an invalid-opcode exception—#UD—if CPL > 0) is determined by the setting of the "MWAIT exiting" VM-execution control:

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 5.4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

  - If the "MWAIT exiting" VM-execution control is 1, MWAIT causes a VM exit.
  - If the "MWAIT exiting" VM-execution control is 0, MWAIT operates normally if one of the following are true: (1) ECX[0] is 0; (2) RFLAGS.IF = 1; or both of the following are true: (a) the "interrupt-window exiting" VM-execution control is 0; and (b) the logical processor has not recognized a pending virtual interrupt (see Section 29.2.1).
  - If the "MWAIT exiting" VM-execution control is 0, ECX[0] = 1, and RFLAGS.IF = 0, MWAIT does not cause the processor to enter an implementation-dependent optimized state if either the "interrupt-window exiting" VM-execution control is 1 or the logical processor has recognized a pending virtual interrupt; instead, control passes to the instruction following the MWAIT instruction.
- **PCONFIG.** Behavior of the PCONFIG instruction is determined by the setting of the "enable PCONFIG" VM-execution control:
  - If the "enable PCONFIG" VM-execution control is 0, PCONFIG causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
  - If the "enable PCONFIG" VM-execution control is 1, PCONFIG may cause a VM exit as specified in Section 27.1.3; if it does not cause such a VM exit, it operates normally.
- **RDMSR.** Section 27.1.3 identifies when executions of the RDMSR instruction cause VM exits. If such an execution causes neither a fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of ECX:
  - If ECX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR), the value returned by the instruction is determined by the setting of the "use TSC offsetting" VM-execution control:
    - If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_TIME_STAMP_COUNTER MSR.
    - If the control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:
      - If the control is 0, RDMSR loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
      - If the control is 1, RDMSR first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

    The 1-setting of the "use TSC-offsetting" VM-execution control does not affect executions of RDMSR if ECX contains 6E0H (indicating the IA32_TSC_DEADLINE MSR). Such executions return the APIC-timer deadline relative to the actual timestamp counter without regard to the TSC offset.
  - If ECX contains 48H (indicating the IA32_SPEC_CTRL MSR), the value returned by the instruction is determined by the setting of the "virtualize IA32_SPEC_CTRL" VM-execution control:
    - If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_SPEC_CTRL MSR.
    - If the control is 1, the value returned is that of the IA32_SPEC_CTRL shadow field in the VMCS.
  - If ECX is in the range 800H–8FFH (indicating an APIC MSR), instruction behavior may be modified if the "virtualize x2APIC mode" VM-execution control is 1; see Section 31.5.
- **RDMSRLIST.** Behavior of the RDMSRLIST instruction is determined first by the setting of the "enable MSR-list instructions" VM-execution control:
  - If the "enable MSR-list instructions" VM-execution control is 0, RDMSRLIST causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.
  - If the "enable MSR-list instructions" VM-execution control is 1, the instruction causes a general-protection exception (#GP) normally if CPL > 0. Otherwise, its operation depends on the setting of the "use MSR bitmaps" VM-execution control:
    - If the control is 0, the instruction causes a VM exit.

    - If the control is 1, the instruction commences normally, reading one MSR at a time. Reads of certain MSRs are treated specially as described above for RDMSR. In addition, attempts to access specific MSRs may cause VM exits; see Section 27.1.3 for details.

- **RDPID.** Behavior of the RDPID instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:
  - If the "enable RDTSCP" VM-execution control is 0, RDPID causes an invalid-opcode exception (#UD).
  - If the "enable RDTSCP" VM-execution control is 1, RDPID operates normally.
- **RDTSC.** Behavior of the RDTSC instruction is determined by the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:
  - If both controls are 0, RDTSC operates normally.
  - If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:
    - If the control is 0, RDTSC loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
    - If the control is 1, RDTSC first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.
  - If the "RDTSC exiting" VM-execution control is 1, RDTSC causes a VM exit.
- **RDTSCP.** Behavior of the RDTSCP instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:
  - If the "enable RDTSCP" VM-execution control is 0, RDTSCP causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.
  - If the "enable RDTSCP" VM-execution control is 1, treatment is based on the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:
    - If both controls are 0, RDTSCP operates normally.
    - If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:
      - If the control is 0, RDTSCP loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
      - If the control is 1, RDTSCP first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

      In either case, RDTSCP also loads ECX with the value of bits 31:0 of the IA32_TSC_AUX MSR.
    - If the "RDTSC exiting" VM-execution control is 1, RDTSCP causes a VM exit.
- **SMSW.** The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, SMSW reads normally from CR0; if every bit is set in the CR0 guest/host mask, SMSW returns the value of the CR0 read shadow.

  Note the following: (1) for any memory destination or for a 16-bit register destination, only the low 16 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:16 of a register destination are left unchanged); (2) for a 32-bit register destination, only the low 32 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:32 of the destination are cleared); and (3) depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.
- **TPAUSE.** Behavior of the TPAUSE instruction is determined first by the setting of the "enable user wait and pause" VM-execution control:

  - If the “enable user wait and pause” VM-execution control is 0, TPAUSE causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
  - If the “enable user wait and pause” VM-execution control is 1, treatment is based on the setting of the “RDTSC exiting” VM-execution control:
    - If the “RDTSC exiting” VM-execution control is 0, the instruction delays for an amount of time called here the **physical delay**. The physical delay is first computed by determining the **virtual delay** (the time to delay relative to the guest’s timestamp counter).

      If IA32_UMWAIT_CONTROL[31:2] is zero, the virtual delay is the value in EDX:EAX minus the value that RDTSC would return (see above); if IA32_UMWAIT_CONTROL[31:2] is not zero, the virtual delay is the minimum of that difference and AND(IA32_UMWAIT_CONTROL,FFFFFFFCH).

      The physical delay depends upon the settings of the “use TSC offsetting” and “use TSC scaling” VM-execution controls:
      - If either control is 0, the physical delay is the virtual delay.
      - If both controls are 1, the virtual delay is multiplied by 2^{48} (using a shift) to produce a 128-bit integer. That product is then divided by the TSC multiplier to produce a 64-bit integer. The physical delay is that quotient.
    - If the “RDTSC exiting” VM-execution control is 1, TPAUSE causes a VM exit.
- **UMONITOR.** Behavior of the UMONITOR instruction is determined by the setting of the “enable user wait and pause” VM-execution control:
  - If the “enable user wait and pause” VM-execution control is 0, UMONITOR causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
  - If the “enable user wait and pause” VM-execution control is 1, UMONITOR operates normally.
- **UMWAIT.** Behavior of the UMWAIT instruction is determined first by the setting of the “enable user wait and pause” VM-execution control:
  - If the “enable user wait and pause” VM-execution control is 0, UMWAIT causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
  - If the “enable user wait and pause” VM-execution control is 1, treatment is based on the setting of the “RDTSC exiting” VM-execution control:
    - If the “RDTSC exiting” VM-execution control is 0, and if the instruction causes a delay, the amount of time delayed is called here the **physical delay**. The physical delay is first computed by determining the **virtual delay** (the time to delay relative to the guest’s timestamp counter).

      If IA32_UMWAIT_CONTROL[31:2] is zero, the virtual delay is the value in EDX:EAX minus the value that RDTSC would return (see above); if IA32_UMWAIT_CONTROL[31:2] is not zero, the virtual delay is the minimum of that difference and AND(IA32_UMWAIT_CONTROL,FFFFFFFCH).

      The physical delay depends upon the settings of the “use TSC offsetting” and “use TSC scaling” VM-execution controls:
      - If either control is 0, the physical delay is the virtual delay.
      - If both controls are 1, the virtual delay is multiplied by 2^{48} (using a shift) to produce a 128-bit integer. That product is then divided by the TSC multiplier to produce a 64-bit integer. The physical delay is that quotient.
    - If the “RDTSC exiting” VM-execution control is 1, UMWAIT causes a VM exit.
- **WRMSR, WRMSRNS.** Section 27.1.3 identifies when an execution of WRMSR or WRMSRNS would cause a VM exit. If such an execution causes neither a fault due to CPL > 0 nor a VM exit, the instruction’s behavior may be modified for certain values of ECX:
  - If ECX contains 48H (indicating the IA32_SPEC_CTRL MSR), instruction behavior depends on the setting of the “virtualize IA32_SPEC_CTRL” VM-execution control:
    - If the control is 0, WRMSR and WRMSRNS operate normally, loading the IA32_SPEC_CTRL MSR with the value in EAX:EDX.

  - If the control is 1, the instruction will attempt to write the IA32_SPEC_CTRL MSR using the instruction's source operand, but it will attempt to modify only those bits in positions corresponding to bits cleared in the IA32_SPEC_CTRL mask field in the VMCS.

    Specifically, the instruction attempts to write the MSR with the following value:

    (MSR_VAL & ISC_MASK) OR (SRC & NOT ISC_MASK),

    where MSR_VAL is the original value of the MSR, ISC_MASK is the IA32_SPEC_CTRL mask, and SRC is the instruction's source operand.

    Any fault that would result from writing that value to the MSR (e.g., due to a reserved-bit violation) occurs normally. Otherwise, the value is written to the MSR.

    Such a write to the MSR will have any side effects that would occur normally had the MSR been written with the value indicated above (including any side effects that may result from writing unchanged values to the masked bits).

    If the write completes without a fault, the unmodified value of the source operand is written to the IA32_SPEC_CTRL shadow field in the VMCS.

— If ECX contains 79H (indicating IA32_BIOS_UPDT_TRIG MSR), no microcode update is loaded, and control passes to the next instruction. This implies that microcode updates cannot be loaded in VMX non-root operation.

— On processors that support Intel PT but which do not allow it to be used in VMX operation, if ECX contains 570H (indicating the IA32_RTIT_CTL MSR), the instruction causes a general-protection exception.[1]

— If ECX contains 808H (indicating the TPR MSR), 80BH (the EOI MSR), 830H (the ICR MSR), or 83FH (the self-IPI MSR), instruction behavior may be modified if the "virtualize x2APIC mode" VM-execution control is 1; see Section 31.5.

- **WRMSRLIST.** Behavior of the WRMSRLIST instruction is determined first by the setting of the "enable MSR-list instructions" VM-execution control:

  — If the "enable MSR-list instructions" VM-execution control is 0, WRMSRLIST causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.

  — If the "enable MSR-list instructions" VM-execution control is 1, the instruction causes a general-protection exception (#GP) normally if CPL > 0. Otherwise, its operation depends on the setting of the "use MSR bitmaps" VM-execution control:

  - If the control is 0, the instruction causes a VM exit.
  - If the control is 1, the instruction commences normally, writing one MSR at a time. Writes to certain MSRs are treated specially as described above for WRMSR and WRMSRNS. In addition, attempts to access specific MSRs may cause VM exits; see Section 27.1.3 for details.

- **XRSTORS.** Behavior of the XRSTORS instruction is determined first by the setting of the "enable XSAVES/XRSTORS" VM-execution control:

  — If the "enable XSAVES/XRSTORS" VM-execution control is 0, XRSTORS causes an invalid-opcode exception (#UD).

  — If the "enable XSAVES/XRSTORS" VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 26.6.21):

  - XRSTORS causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
  - Otherwise, XRSTORS operates normally.

- **XSAVES.** Behavior of the XSAVES instruction is determined first by the setting of the "enable XSAVES/XRSTORS" VM-execution control:

  — If the "enable XSAVES/XRSTORS" VM-execution control is 0, XSAVES causes an invalid-opcode exception (#UD).

1. Software should read the VMX capability MSR IA32_VMX_MISC to determine whether the processor allows Intel PT to be used in VMX operation (see Appendix A.6).

- If the "enable XSAVES/XRSTORS" VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 26.6.21):
  - XSAVES causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
  - Otherwise, XSAVES operates normally.

27.4 OTHER CHANGES IN VMX NON-ROOT OPERATION

Treatments of event blocking, task switches, and certain shadow-stack updates may differ in VMX non-root operation as described in the following sections.

27.4.1 Event Blocking

Event blocking is modified in VMX non-root operation as follows:

- If the "external-interrupt exiting" VM-execution control is 1, RFLAGS.IF does not control the blocking of external interrupts. In this case, an external interrupt that is not blocked for other reasons causes a VM exit (even if RFLAGS.IF = 0).
- If the "external-interrupt exiting" VM-execution control is 1, external interrupts may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).
- If the "NMI exiting" VM-execution control is 1, non-maskable interrupts (NMIs) may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).

27.4.2 Treatment of Task Switches

Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. However, the following checks are performed (in the order indicated), possibly resulting in a fault, before there is any possibility of a VM exit due to task switch:

1. If a task gate is being used, appropriate checks are made on its P bit and on the proper values of the relevant privilege fields. The following cases detail the privilege checks performed:
   a. If CALL, INT *n*, INT1, INT3, INTO, or JMP accesses a task gate in IA-32e mode, a general-protection exception occurs.

   b. If CALL, INT *n*, INT3, INTO, or JMP accesses a task gate outside IA-32e mode, privilege-levels checks are performed on the task gate but, if they pass, privilege levels are not checked on the referenced task-state segment (TSS) descriptor.

   c. If CALL or JMP accesses a TSS descriptor directly in IA-32e mode, a general-protection exception occurs.

   d. If CALL or JMP accesses a TSS descriptor directly outside IA-32e mode, privilege levels are checked on the TSS descriptor.

   e. If a non-maskable interrupt (NMI), an exception, or an external interrupt accesses a task gate in the IDT in IA-32e mode, a general-protection exception occurs.

   f. If a non-maskable interrupt (NMI), an exception other than breakpoint exceptions (#BP) and overflow exceptions (#OF), or an external interrupt accesses a task gate in the IDT outside IA-32e mode, no privilege checks are performed.

   g. If IRET is executed with RFLAGS.NT = 1 in IA-32e mode, a general-protection exception occurs.

   h. If IRET is executed with RFLAGS.NT = 1 outside IA-32e mode, a TSS descriptor is accessed directly and no privilege checks are made.
2. Checks are made on the new TSS selector (for example, that is within GDT limits).
3. The new TSS descriptor is read. (A page fault results if a relevant GDT page is not present).
4. The TSS descriptor is checked for proper values of type (depends on type of task switch), P bit, S bit, and limit.

Only if checks 1–4 all pass (do not generate faults) might a VM exit occur. However, the ordering between a VM exit due to a task switch and a page fault resulting from accessing the old TSS or the new TSS is implementation-specific. Some processors may generate a page fault (instead of a VM exit due to a task switch) if accessing either TSS would cause a page fault. Other processors may generate a VM exit due to a task switch even if accessing either TSS would cause a page fault.

If an attempt at a task switch through a task gate in the IDT causes an exception (before generating a VM exit due to the task switch) and that exception causes a VM exit, information about the event whose delivery that accessed the task gate is recorded in the IDT-vectoring information fields and information about the exception that caused the VM exit is recorded in the VM-exit interruption-information fields. See Section 29.2. The fact that a task gate was being accessed is not recorded in the VMCS.

If an attempt at a task switch through a task gate in the IDT causes VM exit due to the task switch, information about the event whose delivery accessed the task gate is recorded in the IDT-vectoring fields of the VMCS. Since the cause of such a VM exit is a task switch and not an interruption, the valid bit for the VM-exit interruption information field is 0. See Section 29.2.

27.4.3 Shadow-Stack Updates

As noted in Section 18.2.3, “Supervisor Shadow Stack Token,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, a switch of shadow stack may occur as part of IDT event delivery or an execution of far CALL that changes the CPL, or as part of IDT event delivery that uses the interrupt stack table (IST).

As part of the shadow-stack switch, the processor gains exclusive access to the new stack through manipulation of the **supervisor shadow stack token** located at the base of the new shadow stack. The processor reads the token and, among other things, confirms that bit 0 of the token (its **busy bit**) is 0. If the busy bit is already 1, the transition (event delivery or far CALL) cause a general-protection fault and does not complete. If the busy bit is 0, the transition sets the busy bit by writing to the token in memory. (The update is atomic with the original read of the token.)

If the transition commenced with CPL < 3, it will follow the token update by pushing three items on the new shadow stack (for the old values of the CS selector, instruction pointer, and shadow-stack pointer). As noted in Section 18.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, if any of the pushes causes a VM exit, the processor will revert to the old shadow stack and the busy bit in the new shadow stack's token remains set. The new shadow stack is said to be prematurely busy.

If the “prematurely busy shadow stack” VM-exit control is 1, a VM exit that results in a shadow stack becoming prematurely busy will indicate that fact through information saved in the VMCS. See Section 29.2.1.

27.5 FEATURES SPECIFIC TO VMX NON-ROOT OPERATION

Some VM-execution controls support features that are specific to VMX non-root operation. These are the VMX-preemption timer (Section 27.5.1) and the monitor trap flag (Section 27.5.2), translation of guest-physical addresses (Section 27.5.3 and Section 27.5.4), APIC virtualization (Section 27.5.5), VM functions (Section 27.5.6), and virtualization exceptions (Section 27.5.7).

27.5.1 VMX-Preemption Timer

If the last VM entry was performed with the 1-setting of “activate VMX-preemption timer” VM-execution control, the **VMX-preemption timer** counts down (from the value loaded by VM entry; see Section 28.7.4) in VMX non-root operation. When the timer counts down to zero, it stops counting down and a VM exit occurs (see Section 27.2).

The VMX-preemption timer counts down at rate proportional to that of the timestamp counter (TSC). Specifically, the timer counts down by 1 every time bit X in the TSC changes due to a TSC increment. The value of X is in the range 0–31 and can be determined by consulting the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

The VMX-preemption timer operates in the C-states C0, C1, and C2; it also operates in the shutdown and wait-for-SIPI states. If the timer counts down to zero in any state other than the wait-for SIPI state, the logical processor transitions to the C0 C-state and causes a VM exit; the timer does not cause a VM exit if it counts down to zero in the wait-for-SIPI state. The timer is not decremented in C-states deeper than C2.

Treatment of the timer in the case of system management interrupts (SMIs) and system-management mode (SMM) depends on whether the treatment of SMIs and SMM:

- If the default treatment of SMIs and SMM (see Section 33.14) is active, the VMX-preemption timer counts across an SMI to VMX non-root operation, subsequent execution in SMM, and the return from SMM via the RSM instruction. However, the timer can cause a VM exit only from VMX non-root operation. If the timer expires during SMI, in SMM, or during RSM, a timer-induced VM exit occurs immediately after RSM with its normal priority unless it is blocked based on activity state (Section 27.2).
- If the dual-monitor treatment of SMIs and SMM (see Section 33.15) is active, transitions into and out of SMM are VM exits and VM entries, respectively. The treatment of the VMX-preemption timer by those transitions is mostly the same as for ordinary VM exits and VM entries; Section 33.15.2 and Section 33.15.4 detail some differences.

27.5.2 Monitor Trap Flag

The **monitor trap flag** is a debugging feature that causes VM exits to occur on certain instruction boundaries in VMX non-root operation. Such VM exits are called **MTF VM exits**. An MTF VM exit may occur on an instruction boundary in VMX non-root operation as follows:

- If the "monitor trap flag" VM-execution control is 1 and VM entry is injecting a vectored event (see Section 28.6.1), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry.
- If VM entry is injecting a pending MTF VM exit (see Section 28.6.2), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry. This is the case even if the "monitor trap flag" VM-execution control is 0.
- If the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and a pending event (e.g., debug exception or interrupt) is delivered before an instruction can execute, an MTF VM exit is pending on the instruction boundary following delivery of the event (or any nested exception).
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is a REP-prefixed string instruction:
  - If the first iteration of the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).
  - If the first iteration of the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary after that iteration.
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is the XBEGIN instruction. In this case, an MTF VM exit is pending at the fallback instruction address of the XBEGIN instruction. This behavior applies regardless of whether advanced debugging of RTM transactional regions has been enabled (see Section 17.3.7, "RTM-Enabled Debugger Support," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is neither a REP-prefixed string instruction or the XBEGIN instruction:
  - If the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).[1]
  - If the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary following execution of that instruction. If the instruction is INT1, INT3, or INTO, this boundary follows delivery of any software exception. If the instruction is INT *n*, this boundary follows delivery of a software interrupt. If the instruction is HLT, the MTF VM exit will be from the HLT activity state.

No MTF VM exit occurs if another VM exit occurs before reaching the instruction boundary on which an MTF VM exit would be pending (e.g., due to an exception or triple fault).

1. This item includes the cases of an invalid opcode exception—#UD— generated by the UD0, UD1, and UD2 instructions and a BOUND-range exceeded exception—#BR—generated by the BOUND instruction.

An MTF VM exit occurs on the instruction boundary on which it is pending unless a higher priority event takes precedence or the MTF VM exit is blocked due to the activity state:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over MTF VM exits. MTF VM exits take priority over debug-trap exceptions and lower priority events.
- No MTF VM exit occurs if the processor is in either the shutdown activity state or wait-for-SIPI activity state. If a non-maskable interrupt subsequently takes the logical processor out of the shutdown activity state without causing a VM exit, an MTF VM exit is pending after delivery of that interrupt.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 41.2 for details.

27.5.3 Translation of Guest-Physical Addresses Using EPT

The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain physical addresses are treated as guest-physical addresses and are not used to access memory directly. Instead, guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.

Details of the EPT mechanism are given in Section 30.3.

27.5.4 Translation of Guest-Physical Addresses Used by Intel Processor Trace

As described in Chapter 34, Intel® Processor Trace (Intel PT) captures information about software execution using dedicated hardware facilities.

Intel PT can be configured so that the trace output is written to memory using physical addresses. For example, when the ToPA (table of physical addresses) output mechanism is used, the IA32_RTIT_OUTPUT_BASE MSR contains the physical address of the base of the current ToPA. Each entry in that table contains the physical address of an output region in memory. When an output region becomes full, the ToPA output mechanism directs subsequent trace output to the next output region as indicated in the ToPA.

When the "Intel PT uses guest physical addresses" VM-execution control is 1, the logical processor treats the addresses used by Intel PT (the output addresses as well as those used to discover the output addresses) as guest-physical addresses, translating to physical addresses using EPT before trace output is written to memory.

Translating these addresses through EPT implies that the trace-output mechanism may cause EPT violations and VM exits; details are provided in Section 27.5.4.1. Section 27.5.4.2 describes a mechanism that ensures that these VM exits do not cause loss of trace data.

27.5.4.1 Guest-Physical Address Translation for Intel PT: Details

When the "Intel PT uses guest physical addresses" VM-execution control is 1, the addresses used by Intel PT are treated as guest-physical addresses and translated using EPT. These addresses include the addresses of the output regions as well as the addresses of the ToPA entries that contain the output-region addresses.

Translation of accesses by the trace-output process may result in EPT violations or EPT misconfigurations (Section 30.3.3), resulting in VM exits. EPT violations resulting for the trace-output process always cause VM exits and are never converted to virtualization exceptions (Section 27.5.7.1).

If no EPT violation or EPT misconfiguration occurs and if page-modification logging (Section 30.3.6) is enabled, the address of an output region may be added to the page-modification log. If the log is full, a page-modification log-full event occurs, resulting in a VM exit.

If the "virtualize APIC accesses" VM-execution control is 1, a guest-physical address used by the trace-output process may be translated to an address on the APIC-access page. In this case, the access by the trace-output process causes an APIC-access VM exit as discussed in Section 31.4.6.1.

27.5.4.2 Trace-Address Pre-Translation (TAPT)

Because it buffers trace data produced by Intel PT before it is written to memory, the processor ensures that buffered data is not lost when a VM exit disables Intel PT. Specifically, the processor ensures that there is sufficient space left in the current output page for the buffered data. If this were not done, buffered trace data could be lost and the resulting trace corrupted.

To prevent the loss of buffered trace data, the processor uses a mechanism called **trace-address pre-translation** (**TAPT**). With TAPT, the processor translates using EPT the guest-physical address of the current output region before that address would be used to write buffered trace data to memory.

Because of TAPT, no translation (and thus no EPT violation) occurs at the time output is written to memory; the writes to memory use translations that were cached as part of TAPT. (The details given in Section 27.5.4.1 apply to TAPT.) TAPT ensures that, if a write to the output region would cause an EPT violation, the resulting VM exit is delivered at the time of TAPT, before the region would be used. This allows software to resolve the EPT violation at that time and ensures that, when it is necessary to write buffered trace data to memory, that data will not be lost due to an EPT violation.

TAPT (and resulting VM exits) may occur at any of the following times:

- When software in VMX non-root operation enables tracing by loading the IA32_RTIT_CTL MSR to set the TraceEn bit, using the WRMSR instruction or the XRSTORS instruction.

  Any VM exit resulting from TAPT in this case is trap-like: the WRMSR or XRSTORS completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).
- At an instruction boundary when one output region becomes full and Intel PT transitions to the next output region.

  VM exits resulting from TAPT in this case take priority over any pending debug exceptions. Such a VM exit will save information about such exceptions in the guest-state area of the VMCS.
- As part of a VM entry that enables Intel PT. See Section 28.5 for details.

TAPT may translate not only the guest-physical address of the current output region but those of subsequent output regions as well. (Doing so may provide better protection of trace data.) This implies that any VM exits resulting from TAPT may result from the translation of output-region addresses other than that of the current output region.

27.5.5 APIC Virtualization

APIC virtualization is a collection of features that can be used to support the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC). When APIC virtualization is enabled, the processor emulates many accesses to the APIC, tracks the state of the virtual APIC, and delivers virtual interrupts — all in VMX non-root operation without a VM exit.

Details of the APIC virtualization are given in Chapter 31.

27.5.6 VM Functions

A **VM function** is an operation provided by the processor that can be invoked from VMX non-root operation without a VM exit. VM functions are enabled and configured by the settings of different fields in the VMCS. Software in VMX non-root operation invokes a VM function with the **VMFUNC** instruction; the value of EAX selects the specific VM function being invoked.

Section 27.5.6.1 explains how VM functions are enabled. Section 27.5.6.2 specifies the behavior of the VMFUNC instruction. Section 27.5.6.3 describes a specific VM function called **EPTP switching**.

27.5.6.1 Enabling VM Functions

Software enables VM functions generally by setting the “enable VM functions” VM-execution control. A specific VM function is enabled by setting the corresponding VM-function control.

Suppose, for example, that software wants to enable EPTP switching (VM function 0; see Section 26.6.14).To do so, it must set the "activate secondary controls" VM-execution control (bit 31 of the primary processor-based VM-execution controls), the "enable VM functions" VM-execution control (bit 13 of the secondary processor-based VM-execution controls) and the "EPTP switching" VM-function control (bit 0 of the VM-function controls).

27.5.6.2 General Operation of the VMFUNC Instruction

The VMFUNC instruction causes an invalid-opcode exception (#UD) if the "enable VM functions" VM-execution controls is 0[1] or the value of EAX is greater than 63 (only VM functions 0–63 can be enable). Otherwise, the instruction causes a VM exit if the bit at position EAX is 0 in the VM-function controls (the selected VM function is not enabled). If such a VM exit occurs, the basic exit reason used is 59 (3BH), indicating "VMFUNC", and the length of the VMFUNC instruction is saved into the VM-exit instruction-length field. If the instruction causes neither an invalid-opcode exception nor a VM exit due to a disabled VM function, it performs the functionality of the VM function specified by the value in EAX.

Individual VM functions may perform additional fault checking (e.g., one might cause a general-protection exception if CPL > 0). In addition, specific VM functions may include checks that might result in a VM exit. If such a VM exit occurs, VM-exit information is saved as described in the previous paragraph. The specification of a VM function may indicate that additional VM-exit information is provided.

The specific behavior of the EPTP-switching VM function (including checks that result in VM exits) is given in Section 27.5.6.3.

27.5.6.3 EPTP Switching

EPTP switching is VM function 0. This VM function allows software in VMX non-root operation to load a new value for the EPT pointer (EPTP), thereby establishing a different EPT paging-structure hierarchy (see Section 30.3 for details of the operation of EPT). Software is limited to selecting from a list of potential EPTP values configured in advance by software in VMX root operation.

Specifically, the value of ECX is used to select an entry from the EPTP list, the 4-KByte structure referenced by the EPTP-list address (see Section 26.6.14; because this structure contains 512 8-Byte entries, VMFUNC causes a VM exit if ECX ≥ 512). The EPTP value in the selected entry is evaluated to determine whether it is valid for EPTP switching: a value is valid if (1) it is the same as the current EPTP value in bits 5:3 (these bits specify the EPT page-walk length); and (2) it would not cause VM entry to fail (see Section 28.2.1.1). If the value is invalid, a VM exit occurs. Otherwise, the value is stored in the EPTP field of the current VMCS and is used for subsequent accesses using guest-physical addresses. The following pseudocode provides details:

```
IF ECX ≥ 512
    THEN VM exit;
    ELSE
        tent_EPTP := 8 bytes from EPTP-list address + 8 * ECX;
        IF tent_EPTP is not a valid EPTP value (would cause VM entry to fail if in EPTP or change EPT page-walk length)
            THEN VM exit;
            ELSE
                write tent_EPTP to the EPTP field in the current VMCS;
                use tent_EPTP as the new EPTP value for address translation;
                IF processor supports the 1-setting of the "EPT-violation #VE" VM-execution control
                    THEN
                        write ECX[15:0] to EPTP-index field in current VMCS;
                        use ECX[15:0] as EPTP index for subsequent EPT-violation virtualization exceptions (see Section 27.5.7.2);
                FI;
        FI;
FI;
```

Execution of the EPTP-switching VM function does not modify the state of any registers; no flags are modified.

1. "Enable VM functions" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "enable VM functions" VM-execution control were 0. See Section 26.6.2.

If the "Intel PT uses guest physical addresses" VM-execution control is 1 and IA32_RTIT_CTL.TraceEn = 1, any execution of the EPTP-switching VM function causes a VM exit.[1]

As noted in Section 27.5.6.2, an execution of the EPTP-switching VM function that causes a VM exit (as specified above), uses the basic exit reason 59, indicating "VMFUNC". The length of the VMFUNC instruction is saved into the VM-exit instruction-length field. No additional VM-exit information is provided.

An execution of VMFUNC loads EPTP from the EPTP list (and thus does not cause a fault or VM exit) is called an **EPTP-switching VMFUNC**. After an EPTP-switching VMFUNC, control passes to the next instruction. The logical processor starts creating and using guest-physical and combined mappings associated with the new value of bits 51:12 of EPTP; the combined mappings created and used are associated with the current VPID and PCID (these are not changed by VMFUNC).[2] If the "enable VPID" VM-execution control is 0, an EPTP-switching VMFUNC invalidates combined mappings associated with VPID 0000H (for all PCIDs and for all EPTRTA values, where EPTRTA is the value of bits 51:12 of EPTP).

Because an EPTP-switching VMFUNC may change the translation of guest-physical addresses, it may affect use of the guest-physical address in CR3. The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration due to the translation of that guest-physical address through the new EPT paging structures. The following items provide details that apply if CR0.PG = 1:

- If 32-bit paging or 4-level paging[3] is in use (either CR4.PAE = 0 or IA32_EFER.LMA = 1), the next memory access with a linear address uses the translation of the guest-physical address in CR3 through the new EPT paging structures. As a result, this access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.
- If PAE paging is in use (CR4.PAE = 1 and IA32_EFER.LMA = 0), an EPTP-switching VMFUNC **does not** load the four page-directory-pointer-table entries (PDPTEs) from the guest-physical address in CR3. The logical processor continues to use the four guest-physical addresses already present in the PDPTEs. The guest-physical address in CR3 is not translated through the new EPT paging structures (until some operation that would load the PDPTEs).

  The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during the translation of a guest-physical address in any of the PDPTEs. A subsequent memory access with a linear address uses the translation of the guest-physical address in the appropriate PDPTE through the new EPT paging structures. As a result, such an access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.

If an EPTP-switching VMFUNC establishes an EPTP value that enables accessed and dirty flags for EPT (by setting bit 6), subsequent memory accesses may fail to set those flags as specified if there has been no appropriate execution of INVEPT since the last use of an EPTP value that does not enable accessed and dirty flags for EPT (because bit 6 is clear) and that is identical to the new value on bits 51:12.

IF the processor supports the 1-setting of the "EPT-violation #VE" VM-execution control, an EPTP-switching VMFUNC loads the value in ECX[15:0] into to EPTP-index field in current VMCS. Subsequent EPT-violation virtualization exceptions will save this value into the virtualization-exception information area (see Section 27.5.7.2).

27.5.7 Virtualization Exceptions

A **virtualization exception** is a new processor exception. It uses vector 20 and is abbreviated #VE.

A virtualization exception can occur only in VMX non-root operation. Virtualization exceptions occur only with certain settings of certain VM-execution controls. Generally, these settings imply that certain conditions that would normally cause VM exits instead cause virtualization exceptions

In particular, the 1-setting of the "EPT-violation #VE" VM-execution control causes some EPT violations to generate virtualization exceptions instead of VM exits. Section 27.5.7.1 provides the details of how the processor determines whether an EPT violation causes a virtualization exception or a VM exit.

1. Such a VM exit ensures the proper recording of trace data that might otherwise be lost during the change of EPT paging-structure hierarchy. Software handling the VM exit can change emulate the VM function and then resume the guest.
2. If the "enable VPID" VM-execution control is 0, the current VPID is 0000H; if CR4.PCIDE = 0, the current PCID is 000H.
3. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

When the processor encounters a virtualization exception, it saves information about the exception to the virtualization-exception information area; see Section 27.5.7.2.

After saving virtualization-exception information, the processor delivers a virtualization exception as it would any other exception; see Section 27.5.7.3 for details.

27.5.7.1 Convertible EPT Violations

If the "EPT-violation #VE" VM-execution control is 0 (e.g., on processors that do not support this feature), EPT violations always cause VM exits. If instead the control is 1, certain EPT violations may be converted to cause virtualization exceptions instead; such EPT violations are **convertible**.

The values of certain EPT paging-structure entries determine which EPT violations are convertible. Specifically, bit 63 of certain EPT paging-structure entries may be defined to mean **suppress #VE**:

- If bits 2:0 of an EPT paging-structure entry are all 0, the entry is not **present**.[1] If the processor encounters such an entry while translating a guest-physical address, it causes an EPT violation. The EPT violation is convertible if and only if bit 63 of the entry is 0.
- If an EPT paging-structure entry is present, the following cases apply:
  - If the value of the EPT paging-structure entry is not supported, the entry is **misconfigured**. If the processor encounters such an entry while translating a guest-physical address, it causes an EPT misconfiguration (not an EPT violation). EPT misconfigurations always cause VM exits.
  - If the value of the EPT paging-structure entry is supported, the following cases apply:
    - If bit 7 of the entry is 1, or if the entry is an EPT PTE, the entry maps a page. If the processor uses such an entry to translate a guest-physical address, and if an access to that address causes an EPT violation, the EPT violation is convertible if and only if bit 63 of the entry is 0.
    - If bit 7 of the entry is 0 and the entry is not an EPT PTE, the entry references another EPT paging structure. The processor does not use the value of bit 63 of the entry to determine whether any subsequent EPT violation is convertible.

If an access to a guest-physical address causes an EPT violation, bit 63 of exactly one of the EPT paging-structure entries used to translate that address is used to determine whether the EPT violation is convertible: either a entry that is not present (if the guest-physical address does not translate to a physical address) or an entry that maps a page (if it does).

A convertible EPT violation instead causes a virtualization exception if the following all hold:

- CR0.PE = 1;
- the logical processor is not in the process of delivering an event through the IDT;
- the EPT violation did not cause a shadow stack to become prematurely busy (see Section 27.4.3);
- the EPT violation does not result from the output process of Intel Processor Trace (Section 27.5.4); and
- the 32 bits at offset 4 in the virtualization-exception information area are all 0.

Delivery of virtualization exceptions writes the value FFFFFFFFH to offset 4 in the virtualization-exception information area (see Section 27.5.7.2). Thus, once a virtualization exception occurs, another can occur only if software clears this field.

27.5.7.2 Virtualization-Exception Information

Virtualization exceptions save data into the virtualization-exception information area (see Section 26.6.20). Table 27-1 enumerates the data saved and the format of the area.

A VMM may allow guest software to access the virtualization-exception information area. If it does, the guest software may modify that memory (e.g., to clear the 32-bit value at offset 4; see Section 27.5.7.1). (This is an exception to the general requirement given in Section 26.11.4.)

1. If the "mode-based execute control for EPT" VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 **or bit 10** is 1.

Table 27-1. Format of the Virtualization-Exception Information Area

Byte Offset	Contents
0	The 32-bit value that would have been saved into the VMCS as an exit reason had a VM exit occurred instead of the virtualization exception. For EPT violations, this value is 48 (00000030H)
4	FFFFFFFFH
8	The 64-bit value that would have been saved into the VMCS as an exit qualification had a VM exit occurred instead of the virtualization exception
16	The 64-bit value that would have been saved into the VMCS as a guest-linear address had a VM exit occurred instead of the virtualization exception
24	The 64-bit value that would have been saved into the VMCS as a guest-physical address had a VM exit occurred instead of the virtualization exception
32	The current 16-bit value of the EPTP index VM-execution control (see Section 26.6.20 and Section 27.5.6.3)

27.5.7.3 Delivery of Virtualization Exceptions

After saving virtualization-exception information, the processor treats a virtualization exception as it does other exceptions:

- If bit 20 (#VE) is 1 in the exception bitmap in the VMCS, a virtualization exception causes a VM exit (see below). If the bit is 0, the virtualization exception is delivered using gate descriptor 20 in the IDT.
- Virtualization exceptions produce no error code. Delivery of a virtualization exception pushes no error code on the stack.
- With respect to double faults, virtualization exceptions have the same severity as page faults. If delivery of a virtualization exception encounters a nested fault that is either contributory or a page fault, a double fault (#DF) is generated. See Chapter 7, “Interrupt 8—Double Fault Exception (#DF)” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

  It is not possible for a virtualization exception to be encountered while delivering another exception (see Section 27.5.7.1).

If a virtualization exception causes a VM exit directly (because bit 20 is 1 in the exception bitmap), information about the exception is saved normally in the VM-exit interruption information field in the VMCS (see Section 29.2.2). Specifically, the event is reported as a hardware exception with vector 20 and no error code. Bit 12 of the field (NMI unblocking due to IRET) is set normally.

If a virtualization exception causes a VM exit indirectly (because bit 20 is 0 in the exception bitmap and delivery of the exception generates an event that causes a VM exit), information about the exception is saved normally in the IDT-vectoring information field in the VMCS (see Section 29.2.4). Specifically, the event is reported as a hardware exception with vector 20 and no error code.

27.5.8 PASID Translation

The ENQCMD and ENQCMDS instructions each performs a 64-byte enqueue store that includes a 20-bit PASID value in bits 19:0. For ENQCMD, the PASID is normally the value of IA32_PASID[19:0], while for ENQCMDS, the PASID is normally read from memory.

If the “PASID translation” VM-execution control is 1, the PASID value identified in the previous paragraph is treated as a **guest PASID**. PASID translation converts this guest PASID to a 20-bit **host PASID**. After this translation, the enqueue store is performed, using the host PASID in place of the guest PASID.

PASID translation is implemented by two hierarchies of data structures (**PASID-translation hierarchies**) configured by a VMM. Guest PASIDs 00000H to 7FFFFH are translated through the low PASID-translation hierarchy, while guest PASIDs 80000 to FFFFFH are translated through the high PASID-translation hierarchy.

The root of each PASID-translation hierarchy is a 4-KByte **PASID directory**. The low PASID directory is located at the low PASID directory address, and the high PASID directory is located at the high PASID directory address (these physical addresses are VM-execution control fields in the VMCS). A PASID directory comprises 512 8-byte entries, each of which has the following format:

- Bit 0 is the entry's present bit. The entry is used only if this bit is 1.
- Bits 11:1 are reserved and must be 0.
- Bits M–1:12 specify the 4-KByte aligned address of a PASID table (see below), where M is the processor's physical-address width.
- Bits 63:M are reserved and must be 0.

A PASID-translation hierarchy also includes up to 512 4-KByte **PASID tables**; each of these is referenced by a PASID directory entry (see above). A PASID table comprises 1024 4-byte entries, each of which has the following format:

- Bits 19:0 are the host PASID specified by the entry.
- Bits 30:20 are reserved and must be 0.
- Bits 31 is the entry's valid bit. The entry is used only if this bit is 1.

When PASID translation is enabled, the guest PASID determined by the instruction (see above) is converted to a host PASID using the following process:

- If bit 19 of guest PASID is clear, the low PASID directory is used; otherwise, the high PASID directory is used.
- Bits 18:10 of the guest PASID select an entry from the PASID directory. A VM exit occurs if the entry's present bit is clear or if any reserved bit is set. Otherwise, bits M:0 of the entry (with bit 0 cleared) contain the physical address of a PASID table.
- Bits 9:0 of the guest PASID select an entry from the PASID table. A VM exit occurs if the entry's valid bit is clear or if any reserved bit is set. Otherwise, bits 19:0 of the entry are the host PASID.

If PASID translation results in a VM exit (due to a present or valid bit being clear, or a reserved bit being set), the instruction does not complete and no enqueue store is performed.

27.6 UNRESTRICTED GUESTS

The first processors to support VMX operation require CR0.PE and CR0.PG to be 1 in VMX operation (see Section 25.8). This restriction implies that guest software cannot be run in unpaged protected mode or in real-address mode. Later processors support a VM-execution control called "unrestricted guest".[1] If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation. Such processors allow guest software to run in unpaged protected mode or in real-address mode. The following items describe the behavior of such software:

- The MOV CR0 instructions does not cause a general-protection exception simply because it would set either CR0.PE and CR0.PG to 0. See Section 27.3 for details.
- A logical processor treats the values of CR0.PE and CR0.PG in VMX non-root operation just as it does outside VMX operation. Thus, if CR0.PE = 0, the processor operates as it does normally in real-address mode (for example, it uses the 16-bit **interrupt table** to deliver interrupts and exceptions). If CR0.PG = 0, the processor operates as it does normally when paging is disabled.
- Processor operation is modified by the fact that the processor is in VMX non-root operation and by the settings of the VM-execution controls just as it is in protected mode or when paging is enabled. Instructions, interrupts, and exceptions that cause VM exits in protected mode or when paging is enabled also do so in real-address mode or when paging is disabled. The following examples should be noted:
  - If CR0.PG = 0, page faults do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, invalid-TSS exceptions do not occur and thus cannot cause VM exits.

1. "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "unrestricted guest" VM-execution control were 0. See Section 26.6.2.

  - If CR0.PE = 0, the following instructions cause invalid-opcode exceptions and do not cause VM exits: INVEPT, INVVPID, LLDT, LTR, SLDT, STR, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON.
- If CR0.PG = 0, each linear address is passed directly to the EPT mechanism for translation to a physical address.[1] The guest memory type passed on to the EPT mechanism is WB (writeback).

1. As noted in Section 28.2.1.1, the “enable EPT” VM-execution control must be 1 if the “unrestricted guest” VM-execution control is 1.

14. Updates to Chapter 29, Volume 3C

Change bars and violet text show changes to Chapter 29 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C:* System Programming Guide, Part 3.

Changes to this chapter:

- Added content for MSR Data in Section 29.2.5, "Information for VM Exits Due to Instruction Execution."

CHAPTER 29 VM EXITS

VM exits occur in response to certain instructions and events in VMX non-root operation as detailed in Section 27.1 through Section 27.2. VM exits perform the following operations:

1. Information about the cause of the VM exit is recorded in the VM-exit information fields and VM-entry control fields are modified as described in Section 29.2.
2. Processor state is saved in the guest-state area (Section 29.3).
3. MSRs may be saved in the VM-exit MSR-store area (Section 29.4). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.
4. The following may be performed in parallel and in any order (Section 29.5):
   - Processor state is loaded based in part on the host-state area and some VM-exit controls. This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM. See Section 33.15.6 for information on how processor state is loaded by such VM exits.
   - Address-range monitoring is cleared.
5. MSRs may be loaded from the VM-exit MSR-load area (Section 29.6). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.

VM exits are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

Section 29.1 clarifies the nature of the architectural state before a VM exit begins. The steps described above are detailed in Section 29.2 through Section 29.6.

Section 33.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, ordinary transitions to SMM are replaced by VM exits to a separate SMM monitor. Called **SMM VM exits**, these are caused by the arrival of an SMI or the execution of VMCALL in VMX root operation. SMM VM exits differ from other VM exits in ways that are detailed in Section 33.15.2.

29.1 ARCHITECTURAL STATE BEFORE A VM EXIT

This section describes the architectural state that exists before a VM exit, especially for VM exits caused by events that would normally be delivered through the IDT. Note the following:

- An exception causes a VM exit **directly** if the bit corresponding to that exception is set in the exception bitmap. A non-maskable interrupt (NMI) causes a VM exit directly if the "NMI exiting" VM-execution control is 1. An external interrupt causes a VM exit directly if the "external-interrupt exiting" VM-execution control is 1. A start-up IPI (SIPI) that arrives while a logical processor is in the wait-for-SIPI activity state causes a VM exit directly. INIT signals that arrive while the processor is not in the wait-for-SIPI activity state cause VM exits directly.
- An exception, NMI, external interrupt, or software interrupt causes a VM exit **indirectly** if it does not do so directly but delivery of the event causes a nested exception, double fault, task switch, APIC access (see Section 31.4), EPT violation, EPT misconfiguration, page-modification log-full event (see Section 30.3.6), or SPP-related event (see Section 30.3.4) that causes a VM exit.
- An event **results** in a VM exit if it causes a VM exit (directly or indirectly).

The following bullets detail when architectural state is and is not updated in response to VM exits:

- If an event causes a VM exit directly, it does not update architectural state as it would have if it had it not caused the VM exit:
  - A debug exception does not update DR6, DR7, or IA32_DEBUGCTL. (Information about the nature of the debug exception is saved in the exit qualification field.)
  - A page fault does not update CR2. (The linear address causing the page fault is saved in the exit-qualification field.)

  - An NMI causes subsequent NMIs to be blocked, but only after the VM exit completes.
  - An external interrupt does not acknowledge the interrupt controller and the interrupt remains pending, unless the "acknowledge interrupt on exit" VM-exit control is 1. In such a case, the interrupt controller is acknowledged and the interrupt is no longer pending.
  - The flags L0 – L3 in DR7 (bit 0, bit 2, bit 4, and bit 6) are not cleared when a task switch causes a VM exit.
  - If a task switch causes a VM exit, none of the following are modified by the task switch: old task-state segment (TSS); new TSS; old TSS descriptor; new TSS descriptor; RFLAGS.NT[1]; or the TR register.
  - No last-exception record is made if the event that would do so directly causes a VM exit.
  - If a machine-check exception causes a VM exit directly, this does not prevent machine-check MSRs from being updated. These are updated by the machine-check event itself and not the resulting machine-check exception.
  - If the logical processor is in an inactive state (see Section 26.4.2) and not executing instructions, some events may be blocked but others may return the logical processor to the active state. Unblocked events may cause VM exits.[2] If an unblocked event causes a VM exit directly, a return to the active state occurs only after the VM exit completes.[3] The VM exit generates any special bus cycle that is normally generated when the active state is entered from that activity state.

    MTF VM exits (see Section 27.5.2 and Section 28.7.8) are not blocked in the HLT activity state. If an MTF VM exit occurs in the HLT activity state, the logical processor returns to the active state only after the VM exit completes. MTF VM exits are blocked the shutdown state and the wait-for-SIPI state.
- If an event causes a VM exit indirectly, the event does update architectural state:
  - A debug exception updates DR6, DR7, and the IA32_DEBUGCTL MSR. No debug exceptions are considered pending.
  - A page fault updates CR2.
  - An NMI causes subsequent NMIs to be blocked before the VM exit commences.
  - An external interrupt acknowledges the interrupt controller and the interrupt is no longer pending.
  - If the logical processor had been in an inactive state, it enters the active state and, before the VM exit commences, generates any special bus cycle that is normally generated when the active state is entered from that activity state.
  - There is no blocking by STI or by MOV SS when the VM exit commences.
  - Processor state that is normally updated as part of delivery through the IDT (CS, RIP, SS, RSP, RFLAGS) is not modified. However, the incomplete delivery of the event may write to the stack.
  - The treatment of last-exception records is implementation dependent:
    - Some processors make a last-exception record when beginning the delivery of an event through the IDT (before it can encounter a nested exception). Such processors perform this update even if the event encounters a nested exception that causes a VM exit (including the case where nested exceptions lead to a triple fault).
    - Other processors delay making a last-exception record until event delivery has reached some event handler successfully (perhaps after one or more nested exceptions). Such processors do not update the last-exception record if a VM exit or triple fault occurs before an event handler is reached.

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.
2. If a VM exit takes the processor from an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.
3. An exception is made if the logical processor had been inactive due to execution of MWAIT; in this case, it is considered to have become active before the VM exit.

- If the "virtual NMIs" VM-execution control is 1, VM entry injects an NMI, and delivery of the NMI causes a nested exception, double fault, task switch, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event, or APIC access that causes a VM exit, virtual-NMI blocking is in effect before the VM exit commences.
- If a VM exit results from a fault, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered during execution of IRET and the "NMI exiting" VM-execution control is 0, any blocking by NMI is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 29.2.3.
- If a VM exit results from a fault, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered during execution of IRET and the "virtual NMIs" VM-execution control is 1, virtual-NMI blocking is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 29.2.3.
- Suppose that a VM exit is caused directly by an x87 FPU Floating-Point Error (#MF) or by any of the following events if the event was unblocked due to (and given priority over) an x87 FPU Floating-Point Error: an INIT signal, an external interrupt, an NMI, an SMI; or a machine-check exception. In these cases, there is no blocking by STI or by MOV SS when the VM exit commences.
- Normally, a last-branch record may be made when an event is delivered through the IDT. However, if such an event results in a VM exit before delivery is complete, no last-branch record is made.
- If machine-check exception results in a VM exit, processor state is suspect and may result in suspect state being saved to the guest-state area. A VM monitor should consult the RIPV and EIPV bits in the IA32_MCG_STATUS MSR before resuming a guest that caused a VM exit resulting from a machine-check exception.
- If a VM exit results from a fault, APIC access (see Section 31.4), EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered while executing an instruction, data breakpoints due to that instruction may have been recognized and information about them may be saved in the pending debug exceptions field (unless the VM exit clears that field; see Section 29.3.4).
- The following VM exits are considered to happen after an instruction is executed:
  - VM exits resulting from debug traps (single-step, I/O breakpoints, and data breakpoints).
  - VM exits resulting from debug exceptions (data breakpoints) whose recognition was delayed by blocking by MOV SS.
  - VM exits resulting from some machine-check exceptions.
  - Trap-like VM exits due to execution of MOV to CR8 when the "CR8-load exiting" VM-execution control is 0 and the "use TPR shadow" VM-execution control is 1 (see Section 31.3). (Such VM exits can occur only from 64-bit mode and thus only on processors that support Intel 64 architecture.)
  - Trap-like VM exits due to execution of WRMSR when the "use MSR bitmaps" VM-execution control is 1; the value of ECX is in the range 800H–8FFH; and the bit corresponding to the ECX value in write bitmap for low MSRs is 0; and the "virtualize x2APIC mode" VM-execution control is 1. See Section 31.5.
  - VM exits caused by APIC-write emulation (see Section 31.4.3.2) that result from APIC accesses as part of instruction execution.

  For these VM exits, the instruction's modifications to architectural state complete before the VM exit occurs. Such modifications include those to the logical processor's interruptibility state (see Table 26-3). If there had been blocking by MOV SS, POP SS, or STI before the instruction executed, such blocking is no longer in effect.

A VM exit that occurs in enclave mode sets bit 27 of the exit-reason field and bit 4 of the guest interruptibility-state field. Before such a VM exit is delivered, an Asynchronous Enclave Exit (AEX) occurs (see Chapter 38, "Enclave Exiting Events"). An AEX modifies architectural state (Section 38.3). In particular, the processor establishes the following architectural state as indicated:

- The following bits in RFLAGS are cleared: CF, PF, AF, ZF, SF, OF, and RF.
- FS and GS are restored to the values they had prior to the most recent enclave entry.
- RIP is loaded with the AEP of interrupted enclave thread.
- RSP is loaded from the URSP field in the enclave's state-save area (SSA).

29.2 RECORDING VM-EXIT INFORMATION AND UPDATING VM-ENTRY CONTROL FIELDS

VM exits begin by recording information about the nature of and reason for the VM exit in the VM-exit information fields. Section 29.2.1 to Section 29.2.5 detail the use of these fields.

In addition to updating the VM-exit information fields, the valid bit (bit 31) is cleared in the VM-entry interruption-information field. If bit 5 of the IA32_VMX_MISC MSR (index 485H) is read as 1 (see Appendix A.6), the value of IA32_EFER.LMA is stored into the "IA-32e mode guest" VM-entry control.[1]

29.2.1 Basic VM-Exit Information

Section 26.9.1 defines the basic VM-exit information fields. The following items detail their use.

- **Exit reason.**
  - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM exit. Appendix C lists the numbers used and their meaning.
  - Bit 25 is set if the "prematurely busy shadow stack" VM-exit control is 1 and the VM exit caused a shadow stack become prematurely busy (see Section 27.4.3). Otherwise, the bit is cleared.
  - Bit 26 of this field is set to 1 if the VM exit occurred after assertion of a bus lock while the "VMM bus-lock detection" VM-execution control was 1. Such VM exits include those that occur due to the 1-setting of that control as well as others that might occur during execution of an instruction that asserted a bus lock.
  - Bit 27 of this field is set to 1 if the VM exit occurred while the logical processor was in enclave mode.

    Such VM exits include those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.

    A VM exit also sets this bit if it is incident to delivery of an event injected by VM entry and the guest interruptibility-state field indicates an enclave interruption (bit 4 of the field is 1).
  - The remainder of the field (bits 31:28 and bits 24:16) is cleared to 0 (certain SMM VM exits may set some of these bits; see Section 33.15.2.3).[2]
- **Exit qualification.** This field is saved for VM exits due to the following causes: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); system-management interrupts (SMIs) that arrive immediately after the execution of I/O instructions; task switches; INVEPT; INVLPG; INVPCID; INVVPID; LGDT; LIDT; LLDT; LTR; RDMSRLIST; SGDT; SIDT; SLDT; STR; VMCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; WBINVD; WBNOINVD; WRMSR; WRMSRLIST; WRMSRNS; XRSTORS; XSAVES; control-register accesses; MOV DR; I/O instructions; MWAIT; accesses to the APIC-access page (see Section 31.4); EPT violations (see Section 30.3.3.2); EOI virtualization (see Section 31.1.4); APIC-write emulation (see Section 31.4.3.3); page-modification log full (see Section 30.3.6); SPP-related events (see Section 30.3.4); and instruction timeout (see Section 27.2). For all other VM exits, this field is cleared. The following items provide details:
  - For a debug exception, the exit qualification contains information about the debug exception. The information has the format given in Table 29-1.

Table 29-1. Exit Qualification for Debug Exceptions

Bit Position(s)	Contents
3:0	B3 – B0. When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if its corresponding enabling bit in DR7 is not set.
10:4	Not currently defined.

1. Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the "unrestricted guest" VM-execution control.
2. Bit 31 of this field is set on certain VM-entry failures; see Section 28.8.

Table 29-1. Exit Qualification for Debug Exceptions (Contd.)

Bit Position(s)	Contents
11	BLD. When set, this bit indicates that a bus lock was asserted while OS bus-lock detection was enabled and CPL > 0 (see Section 19.3.1.6 ("OS Bus-Lock Detection")).[1]
12	Not currently defined.
13	BD. When set, this bit indicates that the cause of the debug exception is "debug register access detected."
14	BS. When set, this bit indicates that the cause of the debug exception is either the execution of a single instruction (if RFLAGS.TF = 1 and IA32_DEBUGCTL.BTF = 0) or a taken branch (if RFLAGS.TF = DEBUGCTL.BTF = 1).
15	Not currently defined.
16	RTM. When set, this bit indicates that a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 17.3.7, "RTM-Enabled Debugger Support," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).[2]
63:17	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

NOTES:

1. In general, the format of this field matches that of DR6. However, DR6 **clears** bit 11 to indicate detection of a bus lock, while this field **sets** the bit to indicate that condition.
2. In general, the format of this field matches that of DR6. However, DR6 **clears** bit 16 to indicate an RTM-related exception, while this field **sets** the bit to indicate that condition.

— For a page-fault exception, the exit qualification contains the linear address that caused the page fault. If linear-address masking had been in effect (Section 4.4), the address recorded reflects the result of that masking and does not contain any masked metadata. On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

  If the page-fault exception occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

— For a start-up IPI (SIPI), the exit qualification contains the SIPI vector information in bits 7:0. Bits 63:8 of the exit qualification are cleared to 0.

— For a task switch, the exit qualification contains details about the task switch, encoded as shown in Table 29-2.

— For INVLPG, the exit qualification contains the linear-address operand of the instruction.

- On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
- If the INVLPG source operand specifies an unusable segment, the linear address specified in the exit qualification will match the linear address that the INVLPG would have used if no VM exit occurred. This address is not architecturally defined and may be implementation-specific.

Table 29-2. Exit Qualification for Task Switches

Bit Position(s)	Contents
15:0	Selector of task-state segment (TSS) to which the guest attempted to switch
29:16	Not currently defined

Table 29-2. Exit Qualification for Task Switches (Contd.)

Bit Position(s)	Contents
31:30	Source of task switch initiation: 0: CALL instruction 1: IRET instruction 2: JMP instruction 3: Task gate in IDT
63:32	Not currently defined. These bits exist only on processors that support Intel 64 architecture.

— For INVEPT, INVPCID, INVVPID, LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, and XSAVES, the exit qualification receives the value of the instruction's displacement field, which is sign-extended to 64 bits if necessary (32 bits on processors that do not support Intel 64 architecture). If the instruction has no displacement (for example, has a register operand), zero is stored into the exit qualification.

  On processors that support Intel 64 architecture, an exception is made for RIP-relative addressing (used only in 64-bit mode). Such addressing causes an instruction to use an address that is the sum of the displacement field and the value of RIP that references the following instruction. In this case, the exit qualification is loaded with the sum of the displacement field and the appropriate RIP value.

  In all cases, bits of this field beyond the instruction's address size are undefined. For example, suppose that the address-size field in the VM-exit instruction-information field (see Section 26.9.4 and Section 29.2.5) reports an *n*-bit address size. Then bits 63:*n* (bits 31:*n* on processors that do not support Intel 64 architecture) of the instruction displacement are undefined.

— For a control-register access, the exit qualification contains information about the access and has the format given in Table 29-3.

— For MOV DR, the exit qualification contains information about the instruction and has the format given in Table 29-4.

— For an I/O instruction, the exit qualification contains information about the instruction and has the format given in Table 29-5.

— For MWAIT, the exit qualification contains a value that indicates whether address-range monitoring hardware was armed. The exit qualification is set either to 0 (if address-range monitoring hardware is not armed) or to 1 (if address-range monitoring hardware is armed).

— For RDMSRLIST and WRMSRLIST, the exit qualification depends on the setting of the "use MSR bitmaps" VM-execution control. If the control is 0, the exit qualification is zero. If the control is 1, the exit qualification is the index of the MSR whose access caused the VM exit (see Section 27.1.3).

— WBINVD and WBNOINVD use the same basic exit reason (see Appendix C). For WBINVD, the exit qualification is 0, while for WBNOINVD it is 1.

— WRMSR and WRMSRNS use the same basic exit reason (see Appendix C). For WRMSR, the exit qualification is 0, while for WRMSRNS it is 1.

— For an APIC-access VM exit resulting from a linear access or a guest-physical access to the APIC-access page (see Section 31.4), the exit qualification contains information about the access and has the format given in Table 29-6.[1]

  If the access to the APIC-access page occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

  Such a VM exit that set bits 15:12 of the exit qualification to 0000b (data read during instruction execution) or 0001b (data write during instruction execution) set bit 12—which distinguishes data read from data

1. The exit qualification is undefined if the access was part of the logging of a branch record or a processor-event-based-sampling (PEBS) record to the DS save area. It is recommended that software configure the paging structures so that no address in the DS save area translates to an address on the APIC-access page.

write—to that which would have been stored in bit 1—W/R—of the page-fault error code had the access caused a page fault instead of an APIC-access VM exit. This implies the following:

- For an APIC-access VM exit caused by the CLFLUSH and CLFLUSHOPT instructions, the access type is "data read during instruction execution."
- For an APIC-access VM exit caused by the ENTER instruction, the access type is "data write during instruction execution."

Table 29-3. Exit Qualification for Control-Register Accesses

Bit Positions	Contents
3:0	Number of control register (0 for CLTS and LMSW). Bit 3 is always 0 on processors that do not support Intel 64 architecture as they do not support CR8.
5:4	Access type: 0 = MOV to CR 1 = MOV from CR 2 = CLTS 3 = LMSW
6	LMSW operand type: 0 = register 1 = memory For CLTS and MOV CR, cleared to 0
7	Not currently defined
11:8	For MOV CR, the general-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) For CLTS and LMSW, cleared to 0
15:12	Not currently defined
31:16	For LMSW, the LMSW source data For CLTS and MOV CR, cleared to 0
63:32	Not currently defined. These bits exist only on processors that support Intel 64 architecture.

- For an APIC-access VM exit caused by the MASKMOVQ instruction or the MASKMOVDQU instruction, the access type is "data write during instruction execution."
- For an APIC-access VM exit caused by the MONITOR instruction, the access type is "data read during instruction execution."
- For an APIC-access VM exit caused directly by an access to a linear address in the DS save area (BTS or PEBS), the access type is "linear access for monitoring."
- For an APIC-access VM exit caused by a guest-physical access performed for an access to the DS save area (e.g., to access a paging structure to translate a linear address), the access type is "guest-physical access for monitoring or trace."

- For an APIC-access VM exit caused by trace-address pre-translation (TAPT) when the “Intel PT uses guest physical addresses” VM-execution control is 1, the access type is “guest-physical access for monitoring or trace.”

Such a VM exit stores 1 for bit 31 for IDT-vectoring information field (see Section 29.2.4) if and only if it sets bits 15:12 of the exit qualification to 0011b (linear access during event delivery) or 1010b (guest-physical access during event delivery).

See Section 31.4.4 for further discussion of these instructions and APIC-access VM exits.

For APIC-access VM exits resulting from physical accesses to the APIC-access page (see Section 31.4.6), the exit qualification is undefined.

— For an EPT violation, the exit qualification contains information about the access causing the EPT violation and has the format given in Table 29-7.

As noted in that table, the format and meaning of the exit qualification depends on the setting of the “mode-based execute control for EPT” VM-execution control and whether the processor supports advanced VM-exit information for EPT violations.[1]

An EPT violation that occurs during as a result of execution of a read-modify-write operation sets bit 1 (data write). Whether it also sets bit 0 (data read) is implementation-specific and, for a given implementation, may differ for different kinds of read-modify-write operations.

Table 29-4. Exit Qualification for MOV DR

Bit Position(s)	**Contents**
2:0	Number of debug register
3	Not currently defined
4	Direction of access (0 = MOV to DR; 1 = MOV from DR)
7:5	Not currently defined
11:8	General-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8 –15 = R8 – R15, respectively
63:12	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

1. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).

Table 29-5. Exit Qualification for I/O Instructions

Bit Position(s)	Contents
2:0	Size of access: 0 = 1-byte 1 = 2-byte 3 = 4-byte Other values not used
3	Direction of the attempted access (0 = OUT, 1 = IN)
4	String instruction (0 = not string; 1 = string)
5	REP prefixed (0 = not REP; 1 = REP)
6	Operand encoding (0 = DX, 1 = immediate)
15:7	Not currently defined
31:16	Port number (as specified in DX or in an immediate operand)
63:32	Not currently defined. These bits exist only on processors that support Intel 64 architecture.

Table 29-6. Exit Qualification for APIC-Access VM Exits from Linear Accesses and Guest-Physical Accesses

Bit Position(s)	Contents
11:0	▪ If the APIC-access VM exit is due to a linear access, the offset of access within the APIC page. ▪ Undefined if the APIC-access VM exit is due a guest-physical access
15:12	Access type: 0 = linear access for a data read during instruction execution 1 = linear access for a data write during instruction execution 2 = linear access for an instruction fetch 3 = linear access (read or write) during event delivery 4 = linear access for monitoring 10 = guest-physical access during event delivery 11 = guest-physical access for monitoring or trace 15 = guest-physical access for an instruction fetch or during instruction execution Other values not used
16	This bit is set for certain accesses that are asynchronous to instruction execution and not part of event delivery. These includes guest-physical accesses related to trace output by Intel PT (see Section 27.5.4), accesses related to PEBS on processors with the "EPT-friendly" enhancement (see Section 21.9.5), and accesses that occur during user-interrupt delivery (see Section 8.4.2).
63:17	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

Bit 12 reports "NMI unblocking due to IRET"; see Section 29.2.3.

Bit 16 is set for certain accesses that are asynchronous to instruction execution and not part of event delivery. These include trace-address pre-translation (TAPT) for Intel PT (see Section 27.5.4), accesses related to PEBS on processors with the "EPT-friendly" enhancement (see Section 21.9.5), and accesses as part of user-interrupt delivery (see Section 8.4.2).

— For VM exits caused as part of EOI virtualization (Section 31.1.4), bits 7:0 of the exit qualification are set to vector of the virtual interrupt that was dismissed by the EOI virtualization. Bits above bit 7 are cleared.

  - For APIC-write VM exits (Section 31.4.3.3), bits 11:0 of the exit qualification are set to the page offset of the write access that caused the VM exit.[1] Bits above bit 11 are cleared.
  - For a VM exit due to a page-modification log-full event (Section 30.3.6), bit 12 of the exit qualification reports "NMI unblocking due to IRET" (see Section 29.2.3). Bit 16 is set if the VM exit occurs during TAPT, EPT-friendly PEBS, or user-interrupt delivery. All other bits of the exit qualification are undefined.
  - For a VM exit due to an SPP-related event (Section 30.3.4), bit 11 of the exit qualification indicates the type of event: 0 indicates an SPP misconfiguration and 1 indicates an SPP miss. Bit 12 of the exit qualification reports "NMI unblocking due to IRET" (see Section 29.2.3). Bit 16 is set if the VM exit occurs during TAPT EPT-friendly PEBS, or user-interrupt delivery. All other bits of the exit qualification are undefined.
  - If the "PASID translation" VM-execution control, PASID translation is performed for executions of the ENQCMD and ENQCMDS instructions (see Section 27.5.8). PASID translation may fail, resulting in a VM exit. Such a VM exit saves an exit qualification specified in the following items:
    - For ENQCMD, the exit qualification is IA32_PASID[19:0].
    - For ENQCMDS, the exit qualification contains the low 32 bits of the instruction's source operand (which had been read from memory prior to PASID translation).
  - For a VM exit due to an instruction timeout (Section 27.2), bit 0 indicates (if set) that the context of the virtual machine is invalid and that the VM should not be resumed. Bit 12 of the exit qualification reports "NMI unblocking due to IRET" (see Section 29.2.3). All other bits of the exit qualification are undefined.
- **Guest linear address.** For some VM exits, this field receives a linear address that pertains to the VM exit. The field is set for different VM exits as follows:
  - VM exits due to attempts to execute LMSW with a memory operand. In these cases, this field receives the linear address of that operand. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit. If linear-address masking had been in effect (Section 4.4), the address recorded reflects the result of that masking and does not contain any masked metadata.
  - VM exits due to attempts to execute INS or OUTS for which the relevant segment is usable (if the relevant segment is not usable, the value is undefined). (ES is always the relevant segment for INS; for OUTS, the relevant segment is DS unless overridden by an instruction prefix.) The linear address is the base address of relevant segment plus (E)DI (for INS) or (E)SI (for OUTS). Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit. If linear-address masking had been in effect (Section 4.4), the address recorded is the original address before any masking (and may thus contain any metadata).
  - VM exits due to EPT violations that set bit 7 of the exit qualification (see Table 29-7; these are all EPT violations except those resulting from an attempt to load the PDPTEs as of execution of the MOV CR instruction and those due to TAPT). The linear address may translate to the guest-physical address whose access caused the EPT violation. Alternatively, translation of the linear address may reference a paging-structure entry whose access caused the EPT violation. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit. If linear-address masking had been in effect (Section 4.4), the address recorded reflects the result of that masking and does not contain any masked metadata.

    If the EPT violation occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.
  - VM exits due to SPP-related events. If linear-address masking had been in effect (Section 4.4), the address recorded reflects the result of that masking and does not contain any masked metadata.
  - If the "prematurely busy shadow stack" VM-exit control is 1, certain VM exits (besides those noted above) save the linear address that pertains to the VM exit if the VM exit caused a shadow stack to become prematurely busy (see Section 27.4.3). This is true for VM exits due for these reasons: EPT misconfiguration, page-modification log-full event, and instruction timeout. (A VM exit due to instruction timeout that sets bit 0 of the exit qualification, indicating that VM context is invalid, does not save a valid linear address.) If linear-address masking had been in effect (Section 4.4), the address recorded reflects the result of that masking and does not contain any masked metadata.
  - For all other VM exits, the field is undefined.

1. Execution of WRMSR with ECX = 83FH (self-IPI MSR) can lead to an APIC-write VM exit; the exit qualification for such an APIC-write VM exit is 3F0H.

- **Guest-physical address.** For a VM exit due to an EPT violation, an EPT misconfiguration, or an SPP-related event, this field receives the guest-physical address that caused the EPT violation or EPT misconfiguration. For all other VM exits, the field is undefined.

  If the EPT violation or EPT misconfiguration occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

Table 29-7. Exit Qualification for EPT Violations

Bit Position(s)	Contents
0	Set if the access causing the EPT violation was a data read.[1]
1	Set if the access causing the EPT violation was a data write.[1]
2	Set if the access causing the EPT violation was an instruction fetch.
3	The logical-AND of bit 0 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was readable).[2]
4	The logical-AND of bit 1 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was writeable).[2]
5	The logical-AND of bit 2 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation.[2] If the "mode-based execute control for EPT" VM-execution control is 0, this indicates whether the guest-physical address was executable. If that control is 1, this indicates whether the guest-physical address was executable for supervisor-mode linear addresses.
6	If the "mode-based execute control" VM-execution control is 0, the value of this bit is undefined. If that control is 1, this bit is the logical-AND of bit 10 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation. In this case, it indicates whether the guest-physical address was executable for user-mode linear addresses.[3]
7	Set if the guest linear-address field is valid. The guest linear-address field is valid for all EPT violations except those resulting from an attempt to load the guest PDPTEs as part of the execution of the MOV CR instruction and those due to trace-address pre-translation (TAPT; Section 27.5.4).
8	If bit 7 is 1: ▪ Set if the access causing the EPT violation is to a guest-physical address that is the translation of a linear address. ▪ Clear if the access causing the EPT violation is to a paging-structure entry as part of a page walk or the update of an accessed or dirty bit. Reserved if bit 7 is 0 (cleared to 0).
9	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations,[4] this bit is 0 if the linear address is a supervisor-mode linear address and 1 if it is a user-mode linear address. (If CR0.PG = 0, the translation of every linear address is a user-mode linear address and thus this bit will be 1.) Otherwise, this bit is undefined.
10	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations,[4] this bit is 0 if paging translates the linear address to a read-only page and 1 if it translates to a read/write page. (If CR0.PG = 0, every linear address is read/write and thus this bit will be 1.) Otherwise, this bit is undefined.
11	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations,[4] this bit is 0 if paging translates the linear address to an executable page and 1 if it translates to an execute-disable page. (If CR0.PG = 0, CR4.PAE = 0, or IA32_EFER.NXE = 0, every linear address is executable and thus this bit will be 0.) Otherwise, this bit is undefined.
12	NMI unblocking due to IRET (see Section 29.2.3).

Table 29-7. Exit Qualification for EPT Violations (Contd.)

Bit Position(s)	Contents
13	Set if the access causing the EPT violation was a shadow-stack access.
14	If supervisor shadow-stack control is enabled (by setting bit 7 of EPTP), this bit is the same as bit 60 in the EPT paging-structure entry that maps the page of the guest-physical address of the access causing the EPT violation. Otherwise (or if translation of the guest-physical address terminates before reaching an EPT paging-structure entry that maps a page), this bit is undefined.
15	This bit is set if the EPT violation was caused as a result of guest-paging verification. See Section 30.3.3.2.
16	This bit is set if the access was asynchronous to instruction execution not the result of event delivery. The bit is set if the access is related to trace output by Intel PT (see Section 27.5.4), accesses related to PEBS on processors with the "EPT-friendly" enhancement (see Section 21.9.5), or to user-interrupt delivery (see Section 8.4.2). Otherwise, this bit is cleared.
63:17	Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

NOTES:

1. If accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations (see Section 30.3.3.2). If such an access causes an EPT violation, the processor sets both bit 0 and bit 1 of the exit qualification.
2. Bits 5:3 are cleared to 0 if either (1) any of EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation is not present; or (2) 4-level EPT is in use and the guest-physical address sets any bits in the range 51:48 (see Section 30.3.2).
3. Bit 6 is cleared to 0 if (1) the "mode-based execute control" VM-execution control is 1; and (2) either (a) any of EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation is not present; or (b) 4-level EPT is in use and the guest-physical address sets any bits in the range 51:48 (see Section 30.3.2).
4. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).

29.2.2 Information for VM Exits Due to Vectored Events

Section 26.9.2 defines fields containing information for VM exits due to the following events: exceptions (including those generated by the instructions INT1, INT3, INTO, BOUND, UD0, UD1, and UD2); external interrupts that occur while the "acknowledge interrupt on exit" VM-exit control is 1; and non-maskable interrupts (NMIs).[1] Such VM exits include those that occur on an attempt at a task switch that causes an exception before generating the VM exit due to the task switch that causes the VM exit.

The following items detail the use of these fields:

- **VM-exit interruption information** (format given in Table 26-19). The following items detail how this field is established for VM exits due to these events:
  - For an exception, bits 7:0 receive the exception vector (at most 31). For an NMI, bits 7:0 are set to 2. For an external interrupt, bits 7:0 receive the vector.
  - Bits 10:8 are set to 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 5 (privileged software exception), or 6 (software exception). Hardware exceptions comprise all exceptions except the following:
    - Debug exceptions (#DB) generated by the INT1 instruction; these are privileged software exceptions. (Other debug exceptions are considered hardware exceptions, as are those caused by executions of INT1 in enclave mode.)
    - Breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.)

1. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with value 1 or 3 for n.

BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD0, UD1, and UD2 are hardware exceptions.

- Bit 11 is set to 1 if the VM exit is caused by a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).[1] If bit 11 is set to 1, the error code is placed in the VM-exit interruption error code (see below).
- Bit 12 reports "NMI unblocking due to IRET"; see Section 29.2.3. The value of this bit is undefined if the VM exit is due to a double fault (the interruption type is hardware exception and the vector is 8).
- Bits 30:13 are always set to 0.
- Bit 31 is always set to 1.

For other VM exits (including those due to external interrupts when the "acknowledge interrupt on exit" VM-exit control is 0), the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- **VM-exit interruption error code**.
  - For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the VM-exit interruption-information field, this field receives the error code that would have been pushed on the stack had the event causing the VM exit been delivered normally through the IDT. The EXT bit is set in this field exactly when it would be set normally. For exceptions that occur during the delivery of double fault (if the IDT-vectoring information field indicates a double fault), the EXT bit is set to 1, assuming that (1) that the exception would produce an error code normally (if not incident to double-fault delivery) and (2) that the error code uses the EXT bit (not for page faults, which use a different format).
  - For other VM exits, the value of this field is undefined.

29.2.3 Information About NMI Unblocking Due to IRET

A VM exit may occur during execution of the IRET instruction for reasons including the following: faults, EPT violations, page-modification log-full events, SPP-related events, or instruction timeouts.

An execution of IRET that commences while non-maskable interrupts (NMIs) are blocked will unblock NMIs even if a fault or VM exit occurs; the state saved by such a VM exit will indicate that NMIs were not blocked.

VM exits for the reasons enumerated above provide more information to software by saving a bit called "NMI unblocking due to IRET." This bit is defined if (1) either the "NMI exiting" VM-execution control is 0 or the "virtual NMIs" VM-execution control is 1; (2) the VM exit does not set the valid bit in the IDT-vectoring information field (see Section 29.2.4); and (3) the VM exit is not due to a double fault. In these cases, the bit is defined as follows:

- The bit is 1 if the VM exit resulted from a memory access as part of execution of the IRET instruction and one of the following holds:
  - The "virtual NMIs" VM-execution control is 0 and blocking by NMI (see Table 26-3) was in effect before execution of IRET.
  - The "virtual NMIs" VM-execution control is 1 and virtual-NMI blocking was in effect before execution of IRET.
- The bit is 0 for all other relevant VM exits.

For VM exits due to faults, NMI unblocking due to IRET is saved in bit 12 of the VM-exit interruption-information field (Section 29.2.2). For VM exits due to EPT violations, page-modification log-full events, SPP-related events, and instruction timeouts, NMI unblocking due to IRET is saved in bit 12 of the exit qualification (Section 29.2.1).

(Executions of IRET may also incur VM exits due to APIC accesses and EPT misconfigurations. These VM exits do not report information about NMI unblocking due to IRET.)

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

29.2.4 Information for VM Exits During Event Delivery

Section 26.9.3 defined fields containing information for VM exits that occur while delivering an event through the IDT and as a result of any of the following cases:[1]

- A fault occurs during event delivery and causes a VM exit (because the bit associated with the fault is set to 1 in the exception bitmap).
- A task switch is invoked through a task gate in the IDT. The VM exit occurs due to the task switch only after the initial checks of the task switch pass (see Section 27.4.2).
- Event delivery causes an APIC-access VM exit (see Section 31.4).
- An EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that occurs during event delivery.
- Any of the above VM exits that occur during user-interrupt notification processing (see Section 8.5.2). Such VM exits will be treated as if they occurred during delivery of an external interrupt with the vector UINV.

These fields are used for VM exits that occur during delivery of events injected as part of VM entry (see Section 28.6.1.2).

A VM exit is not considered to occur during event delivery in any of the following circumstances:

- The original event causes the VM exit directly (for example, because the original event is a non-maskable interrupt (NMI) and the “NMI exiting” VM-execution control is 1).
- The original event results in a double-fault exception that causes the VM exit directly.
- The VM exit occurred as a result of fetching the first instruction of the handler invoked by the event delivery.
- The VM exit is caused by a triple fault.
- The original event was a software interrupt (INT *n*) executed in virtual-8086 mode with EFLAGS.IOPL < 3 and the VM exit was due to a general-protection exception (#GP) that occurred because either CR4.VME = 0 or bit *n* of the software interrupt redirection bit map in the TSS is set.

The following items detail the use of these fields:

- IDT-vectoring information (format given in Table 26-20). The following items detail how this field is established for VM exits that occur during event delivery:
  - — If the VM exit occurred during delivery of an exception, bits 7:0 receive the exception vector (at most 31). If the VM exit occurred during delivery of an NMI, bits 7:0 are set to 2. If the VM exit occurred during delivery of an external interrupt, bits 7:0 receive the vector.
  - — Bits 10:8 are set to indicate the type of event that was being delivered when the VM exit occurred: 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 4 (software interrupt), 5 (privileged software interrupt), or 6 (software exception).

    Hardware exceptions comprise all exceptions except the following:[2]

    - Debug exceptions (#DB) generated by the INT1 instruction; these are privileged software exceptions. (Other debug exceptions are considered hardware exceptions, as are those caused by executions of INT1 in enclave mode.)
    - Breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.)

    BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD0, UD1, and UD2 are hardware exceptions.
  - — Bit 11 is set to 1 if the VM exit occurred during delivery of a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in

1. This includes the case in which a VM exit occurs while delivering a software interrupt (INT *n*) through the 16-bit IVT (interrupt vector table) that is used in virtual-8086 mode with virtual-machine extensions (if RFLAGS.VM = CR4.VME = 1).
2. In the following items, INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT *n* with value 1 or 3 for n.

real-address mode (CR0.PE=0).[1] If bit 11 is set to 1, the error code is placed in the IDT-vectoring error code (see below).

— Bit 12 is undefined.

— Bits 30:13 are always set to 0.

— Bit 31 is always set to 1.

For other VM exits, the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- IDT-vectoring error code.
  - — For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the IDT-vectoring information field, this field receives the error code that would have been pushed on the stack by the event that was being delivered through the IDT at the time of the VM exit. The EXT bit is set in this field when it would be set normally.
  - — For other VM exits, the value of this field is undefined.

29.2.5 Information for VM Exits Due to Instruction Execution

Section 26.9.4 defined fields containing information for VM exits that occur due to instruction execution. (The VM-exit instruction length is also used for VM exits that occur during the delivery of a software interrupt or software exception.) The following items detail their use.

- **VM-exit instruction length.** This field is used in the following cases:
  - — For fault-like VM exits due to attempts to execute one of the following instructions that cause VM exits unconditionally (see Section 27.1.2) or based on the settings of VM-execution controls (see Section 27.1.3): CLTS, CPUID, ENCLS, GETSEC, HLT, IN, INS, INVD, INVEPT, INVLPG, INVPCID, INVVPID, LGDT, LIDT, LLDT, LMSW, LOADIWKEY, LTR, MONITOR, MOV CR, MOV DR, MWAIT, OUT, OUTS, PAUSE, PCONFIG, RDMSR, RDPMC, RDRAND, RDSEED, RDTSC, RDTSCP, RSM, SGDT, SIDT, SLDT, STR, TPAUSE, UMWAIT, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON, WBINVD, WBNOINVD, WRMSR, XRSTORS, XSETBV, and XSAVES.[2]
  - — For VM exits due to software exceptions (those generated by executions of INT3 or INTO) or privileged software exceptions (those generated by executions of INT1).
  - — For VM exits due to faults encountered during delivery of a software interrupt, privileged software exception, or software exception.
  - — For VM exits due to attempts to effect a task switch via instruction execution. These are VM exits that produce an exit reason indicating task switch and either of the following:
    - An exit qualification indicating execution of CALL, IRET, or JMP instruction.
    - An exit qualification indicating a task gate in the IDT and an IDT-vectoring information field indicating that the task gate was encountered during delivery of a software interrupt, privileged software exception, or software exception.
  - — For APIC-access VM exits and for VM exits caused by EPT violations, page-modification log-full events, and SPP-related events encountered during delivery of a software interrupt, privileged software exception, or software exception.[3]
  - — For VM exits due executions of VMFUNC that fail because one of the following is true:

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

2. This item applies only to fault-like VM exits. It does not apply to trap-like VM exits following executions of the MOV to CR8 instruction when the “use TPR shadow” VM-execution control is 1 or to those following executions of the WRMSR instruction when the “virtualize x2APIC mode” VM-execution control is 1.

3. The VM-exit instruction-length field is not defined following APIC-access VM exits resulting from physical accesses (see Section 31.4.6) even if encountered during delivery of a software interrupt, privileged software exception, or software exception.

- EAX indicates a VM function that is not enabled (the bit at position EAX is 0 in the VM-function controls; see Section 27.5.6.2).
- EAX = 0 and either ECX ≥ 512 or the value of ECX selects an invalid tentative EPTP value (see Section 27.5.6.3).

In all the above cases, this field receives the length in bytes (1–15) of the instruction (including any instruction prefixes) whose execution led to the VM exit (see the next paragraph for one exception).

The cases of VM exits encountered during delivery of a software interrupt, privileged software exception, or software exception include those encountered during delivery of events injected as part of VM entry (see Section 28.6.1.2). If the original event was injected as part of VM entry, this field receives the value of the VM-entry instruction length.

All VM exits other than those listed in the above items leave this field undefined.

If the VM exit occurred in enclave mode, this field is cleared (none of the previous items apply).

Table 29-8. Format of the VM-Exit Instruction-Information Field as Used for INS and OUTS

Bit Position(s)	Content
6:0	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
14:10	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for VM exits due to execution of INS.
31:18	Undefined.

- **VM-exit instruction information**. For VM exits due to attempts to execute INS, INVEPT, INVPCID, INVVPID, LIDT, LGDT, LLDT, LOADIWKEY, LTR, OUTS, RDRAND, RDSEED, SIDT, SGDT, SLDT, STR, TPAUSE, UMWAIT, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, or XSAVES, this field receives information about the instruction that caused the VM exit. The format of the field depends on the identity of the instruction causing the VM exit:
  - For VM exits due to attempts to execute INS or OUTS, the field has the format is given in Table 29-8.[1]
  - For VM exits due to attempts to execute INVEPT, INVPCID, or INVVPID, the field has the format is given in Table 29-9.
  - For VM exits due to attempts to execute LIDT, LGDT, SIDT, or SGDT, the field has the format is given in Table 29-10.
  - For VM exits due to attempts to execute LLDT, LTR, SLDT, or STR, the field has the format is given in Table 29-11.
  - For VM exits due to attempts to execute RDRAND or RDSEED, the field has the format is given in Table 29-12.

1. The format of the field was undefined for these VM exits on the first processors to support the virtual-machine extensions. Software can determine whether the format specified in Table 29-8 is used by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

  - For VM exits due to attempts to execute TPAUSE or UMWAIT, the field has the format is given in Table 29-13.
  - For VM exits due to attempts to execute VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, or XSAVES, the field has the format is given in Table 29-14.
  - For VM exits due to attempts to execute VMREAD or VMWRITE, the field has the format is given in Table 29-15.
  - For VM exits due to attempts to execute LOADIWKEY, the field has the format is given in Table 29-16.

  For all other VM exits, the field is undefined, unless the VM exit occurred in enclave mode, in which case the field is cleared.
- **I/O RCX, I/O RSI, I/O RDI, I/O RIP**. These fields are undefined except for SMM VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions. See Section 33.15.2.3. Note that, if the VM exit occurred in enclave mode, these fields are all cleared.
- **MSR data**. An execution of WRMSRLIST may cause a VM exit if it would write to an MSR for which the MSR bitmaps do not allow writes (see Section 27.1.3). Such VM exits save the 64-bit data that would have been written to the MSR into this field in the VMCS.

Table 29-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVVPID

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).

Table 29-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVVPID (Contd.)

Bit Position(s)	Content
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for memory instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Reg2 (same encoding as IndexReg above)

Table 29-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
11	Operand size: 0: 16-bit 1: 32-bit Undefined for VM exits from 64-bit mode.
14:12	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)

Table 29-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT (Contd.)

Bit Position(s)	Content
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
29:28	Instruction identity: 0: SGDT 1: SIDT 2: LGDT 3: LIDT
31:30	Undefined.

Table 29-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for register instructions (bit 10 is set).

Table 29-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR (Contd.)

Bit Position(s)	Content
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
29:28	Instruction identity: 0: SLDT 1: STR 2: LLDT 3: LTR
31:30	Undefined.

Table 29-12. Format of the VM-Exit Instruction-Information Field as Used for RDRAND and RDSEED

Bit Position(s)	Content
2:0	Undefined.
6:3	Operand register (destination register): 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)
10:7	Undefined.
12:11	Operand size: 0: 16-bit 1: 32-bit 2: 64-bit The value 3 is not used.
31:13	Undefined.

Table 29-13. Format of the VM-Exit Instruction-Information Field as Used for TPAUSE and UMWAIT

Bit Position(s)	Content
2:0	Undefined.

Table 29-13. Format of the VM-Exit Instruction-Information Field as Used for TPAUSE and UMWAIT (Contd.)

Bit Position(s)	Content
6:3	Operand register (source register): 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)
31:7	Undefined.

Table 29-14. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).

Table 29-14. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES (Contd.)

Bit Position(s)	Content
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Undefined.

Table 29-15. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for register instructions (bit 10 is set).
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).

Table 29-15. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE (Contd.)

Bit Position(s)	Content
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
31:28	Reg2 (same encoding as Reg1 above)

Table 29-16. Format of the VM-Exit Instruction-Information Field as Used for LOADIWKEY

Bit Position(s)	Content
2:0	Undefined.
6:3	Reg1: identifies the first XMM register operand (XMM0–XMM15; values 8–15 are used only on processors that support Intel 64 architecture).
30:7	Undefined.
31:28	Reg2: identifies the second XMM register operand (see above).

29.3 SAVING GUEST STATE

VM exits save certain components of processor state into corresponding fields in the guest-state area of the VMCS (see Section 26.4). On processors that support Intel 64 architecture, the full value of each natural-width field (see Section 26.11.2) is saved regardless of the mode of the logical processor before and after the VM exit.

In general, the state saved is that which was in the logical processor at the time the VM exit commences. See Section 29.1 for a discussion of which architectural updates occur at that time.

Section 29.3.1 through Section 29.3.4 provide details for how various components of processor state are saved. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

29.3.1 Saving Control Registers, Debug Registers, and MSRs

Contents of certain control registers, debug registers, and MSRs are saved as follows:

- The contents of CR0, CR3, CR4, and the IA32_SYSENTER_CS, IA32_SYSENTER_ESP, and IA32_SYSENTER_EIP MSRs are saved into the corresponding fields. Bits 63:32 of the IA32_SYSENTER_CS MSR are not saved. On processors that do not support Intel 64 architecture, bits 63:32 of the IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are not saved.
- If the “save debug controls” VM-exit control is 1, the contents of DR7 and the IA32_DEBUGCTL MSR are saved into the corresponding fields. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always saved data into these fields.
- If the “save IA32_PAT” VM-exit control is 1, the contents of the IA32_PAT MSR are saved into the corresponding field.
- If the “save IA32_EFER” VM-exit control is 1, the contents of the IA32_EFER MSR are saved into the corresponding field.
- If the processor supports either the 1-setting of the “load IA32_BNDCFGS” VM-entry control or that of the “clear IA32_BNDCFGS” VM-exit control, the contents of the IA32_BNDCFGS MSR are saved into the corresponding field.

- If the processor supports either the 1-setting of the "load IA32_RTIT_CTL" VM-entry control or that of the "clear IA32_RTIT_CTL" VM-exit control, the contents of the IA32_RTIT_CTL MSR are saved into the corresponding field.
- If the processor supports the 1-setting of the "load CET" VM-entry control, the contents of the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are saved into the corresponding fields. On processors that do not support Intel 64 architecture, bits 63:32 of these MSRs are not saved.
- If the processor supports either the 1-setting of the "load guest IA32_LBR_CTL" VM-entry control or that of the "clear IA32_LBR_CTL" VM-exit control, the contents of the IA32_LBR_CTL MSR are saved into the corresponding field.
- If the processor supports the 1-setting of the "load PKRS" VM-entry control, the contents of the IA32_PKRS MSR are saved into the corresponding field.
- If a processor supports user interrupts, every VM exit saves UINV into the guest UINV field in the VMCS (bits 15:8 of the field are cleared).
- If the "save IA32_PERF_GLOBAL_CTL" VM-exit control is 1, the contents of the IA32_PERF_GLOBAL_CTL MSR are saved into the corresponding field.
- The value of the SMBASE field is undefined after all VM exits except SMM VM exits. See Section 33.15.2.

29.3.2 Saving Segment Registers and Descriptor-Table Registers

For each segment register (CS, SS, DS, ES, FS, GS, LDTR, or TR), the values saved for the base-address, segment-limit, and access rights are based on whether the register was unusable (see Section 26.4.1) before the VM exit:

- If the register was unusable, the values saved into the following fields are undefined: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in the access-rights field. The following exceptions apply:
  - CS.
    - The base-address and segment-limit fields are saved.
    - The L, D, and G bits are saved in the access-rights field.
  - SS.
    - DPL is saved in the access-rights field.
    - On processors that support Intel 64 architecture, bits 63:32 of the value saved for the base address are always zero.
  - DS and ES. On processors that support Intel 64 architecture, bits 63:32 of the values saved for the base addresses are always zero.
  - FS and GS. The base-address field is saved.
- If the register was not unusable, the values saved into the following fields are those which were in the register before the VM exit: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in access rights.
- Bits 31:17 and 11:8 in the access-rights field are always cleared. Bit 16 is set to 1 if and only if the segment is unusable.

The contents of the GDTR and IDTR registers are saved into the corresponding base-address and limit fields.

29.3.3 Saving RIP, RSP, RFLAGS, and SSP

The contents of the RIP, RSP, RFLAGS, and SSP (shadow-stack pointer) registers are saved as follows:

- The value saved in the RIP field is determined by the nature and cause of the VM exit:
  - If the VM exit occurred in enclave mode, the value saved is the AEP of interrupted enclave thread (the remaining items do not apply).
  - If the VM exit occurs due to by an attempt to execute an instruction that causes VM exits unconditionally or that has been configured to cause a VM exit via the VM-execution controls, the value saved references that instruction.

- If the VM exit is caused by an occurrence of an INIT signal, a start-up IPI (SIPI), or system-management interrupt (SMI), the value saved is that which was in RIP before the event occurred.
- If the VM exit occurs due to the 1-setting of either the "interrupt-window exiting" VM-execution control or the "NMI-window exiting" VM-execution control, the value saved is that which would be in the register had the VM exit not occurred.
- If the VM exit is due to an external interrupt, non-maskable interrupt (NMI), or hardware exception (as defined in Section 29.2.2), the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate,[1] or into the old task-state segment had the event been delivered through a task gate).
- If the VM exit is due to a triple fault, the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate, or into the old task-state segment had the event been delivered through a task gate) had delivery of the double fault not encountered the nested exception that caused the triple fault.
- If the VM exit is due to a software exception (due to an execution of INT3 or INTO) or a privileged software exception (due to an execution of INT1), the value saved references the INT3, INTO, or INT1 instruction that caused that exception.
- Suppose that the VM exit is due to a task switch that was caused by execution of CALL, IRET, or JMP or by execution of a software interrupt (INT *n*), software exception (due to execution of INT3 or INTO), or privileged software exception (due to execution of INT1) that encountered a task gate in the IDT. The value saved references the instruction that caused the task switch (CALL, IRET, JMP, INT *n*, INT3, INTO, INT1).
- Suppose that the VM exit is due to a task switch that was caused by a task gate in the IDT that was encountered for any reason except the direct access by a software interrupt or software exception. The value saved is that which would have been saved in the old task-state segment had the task switch completed normally.
- If the VM exit is due to an execution of MOV to CR8 or WRMSR that reduced the value of bits 7:4 of VTPR (see Section 31.1.1) below that of TPR threshold VM-execution control field (see Section 31.1.2), the value saved references the instruction following the MOV to CR8 or WRMSR.
- If the VM exit was caused by APIC-write emulation (see Section 31.4.3.2) that results from an APIC access as part of instruction execution, the value saved references the instruction following the one whose execution caused the APIC-write emulation.

• The contents of the RSP register are saved into the RSP field.

• With the exception of the resume flag (RF; bit 16), the contents of the RFLAGS register is saved into the RFLAGS field. RFLAGS.RF is saved as follows:

- If the VM exit occurred in enclave mode, the value saved is 0 (the remaining items do not apply).
- If the VM exit is caused directly by an event that would normally be delivered through the IDT, the value saved is that which would appear in the saved RFLAGS image (either that which would be saved on the stack had the event been delivered through a trap or interrupt gate[2] or into the old task-state segment had the event been delivered through a task gate) had the event been delivered through the IDT. See below for VM exits due to task switches caused by task gates in the IDT.
- If the VM exit is caused by a triple fault, the value saved is that which the logical processor would have in RF in the RFLAGS register had the triple fault taken the logical processor to the shutdown state.
- If the VM exit is caused by a task switch (including one caused by a task gate in the IDT), the value saved is that which would have been saved in the RFLAGS image in the old task-state segment (TSS) had the task switch completed normally without exception.
- If the VM exit is caused by an attempt to execute an instruction that unconditionally causes VM exits or one that was configured to do with a VM-execution control, the value saved is 0.[3]

---

1. The reference here is to the full value of RIP before any truncation that would occur had the stack width been only 32 bits or 16 bits.
2. The reference here is to the full value of RFLAGS before any truncation that would occur had the stack width been only 32 bits or 16 bits.

  - For APIC-access VM exits and for VM exits caused by EPT violations, EPT misconfigurations, page-modification log-full events, or SPP-related events, the value saved depends on whether the VM exit occurred during delivery of an event through the IDT:
    - If the VM exit stored 0 for bit 31 for IDT-vectoring information field (because the VM exit did not occur during delivery of an event through the IDT; see Section 29.2.4), the value saved is 1.
    - If the VM exit stored 1 for bit 31 for IDT-vectoring information field (because the VM exit did occur during delivery of an event through the IDT), the value saved is the value that would have appeared in the saved RFLAGS image had the event been delivered through the IDT (see above).
  - For all other VM exits, the value saved is the value RFLAGS.RF had before the VM exit occurred.
- If the processor supports the 1-setting of the "load CET" VM-entry control, the contents of the SSP register are saved into the SSP field.

29.3.4 Saving Non-Register State

Information corresponding to guest non-register state is saved as follows:

- The activity-state field is saved with the logical processor's activity state before the VM exit.[1] See Section 29.1 for details of how events leading to a VM exit may affect the activity state. If the VM exit occurred during user-interrupt notification processing (see Section 8.5.2) and the logical processor would have entered the HLT state following user-interrupt notification processing, the saved activity state is "HLT".
- The interruptibility-state field is saved to reflect the logical processor's interruptibility before the VM exit.
  - See Section 29.1 for details of how events leading to a VM exit may affect this state.
  - VM exits that end outside system-management mode (SMM) save bit 2 (blocking by SMI) as 0 regardless of the state of such blocking before the VM exit.
  - Bit 3 (blocking by NMI) is treated specially if the "virtual NMIs" VM-execution control is 1. In this case, the value saved for this field does not indicate the blocking of NMIs but rather the state of virtual-NMI blocking.
  - Bit 4 (enclave interruption) is set to 1 if the VM exit occurred while the logical processor was in enclave mode.

    Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.

    A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified.
- The pending debug exceptions field is saved as clear for all VM exits except the following:
  - A VM exit caused by an INIT signal, a machine-check exception, or a system-management interrupt (SMI).
  - A VM exit with basic exit reason "TPR below threshold",[2] "virtualized EOI", "APIC write", "monitor trap flag," or "bus-lock detected."
  - A VM exit due to trace-address pre-translation (TAPT; see Section 27.5.4) or due to accesses related to PEBS on processors with the "EPT-friendly" enhancement (see Section 21.9.5). Such VM exits can have basic exit reason "APIC access," "EPT violation," "EPT misconfiguration," "page-modification log full," or "SPP-related event." When due to TAPT or PEBS, these VM exits (with the exception of those due to EPT misconfigurations) set bit 16 of the exit qualification, indicating that they are asynchronous to instruction execution and not part of event delivery.

3. This is true even if RFLAGS.RF was 1 before the instruction was executed. If, in response to such a VM exit, a VM monitor re-enters the guest to re-execute the instruction that caused the VM exit (for example, after clearing the VM-execution control that caused the VM exit), the instruction may encounter a code breakpoint that has already been processed. A VM monitor can avoid this by setting the guest value of RFLAGS.RF to 1 before resuming guest software.

1. If this activity state was an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.

2. This item includes VM exits that occur as a result of certain VM entries (Section 28.7.7).

- VM exits that are not caused by debug exceptions and that occur while there is MOV-SS blocking of debug exceptions.

For VM exits that do not clear the field, the value saved is determined as follows:

- Each of bits 3:0 may be set if it corresponds to a matched breakpoint. This may be true even if the corresponding breakpoint is not enabled in DR7.
- Suppose that a VM exit is due to an INIT signal, a machine-check exception, or an SMI; or that a VM exit has basic exit reason "TPR below threshold" or "monitor trap flag." In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit.

  If the VM exit occurs immediately after VM entry, the value saved may match that which was loaded on VM entry (see Section 28.7.3). Otherwise, the following items apply:

  - Bit 12 (enabled breakpoint) is set to 1 in any of the following cases:
    - If there was at least one matched data or I/O breakpoint that was enabled in DR7.
    - If it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 28.7.3) and the VM exit occurred before those exceptions were either delivered or lost.
    - If the XBEGIN instruction was executed immediately before the VM exit and advanced debugging of RTM transactional regions had been enabled (see Section 17.3.7, "RTM-Enabled Debugger Support," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). (This does not apply to VM exits with basic exit reason "monitor trap flag.")
    - If a bus lock was asserted while CPL > 0 and OS bus-lock detection was enabled.

    In other cases, bit 12 is cleared to 0.
  - Bit 14 (BS) is set if RFLAGS.TF = 1 in either of the following cases:
    - IA32_DEBUGCTL.BTF = 0 and the cause of a pending debug exception was the execution of a single instruction.
    - IA32_DEBUGCTL.BTF = 1 and the cause of a pending debug exception was a taken branch.
  - Bit 16 (RTM) is set if a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions had been enabled. (This does not apply to VM exits with basic exit reason "monitor trap flag.")
- Suppose that a VM exit is due to another reason (but not a debug exception) and occurs while there is MOV-SS blocking of debug exceptions. In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit. If the VM exit occurs immediately after VM entry (no instructions were executed in VMX non-root operation), the value saved may match that which was loaded on VM entry (see Section 28.7.3). Otherwise, the following items apply:
  - Bit 12 (enabled breakpoint) is set to 1 if there was at least one matched data or I/O breakpoint that was enabled in DR7. Bit 12 is also set if it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 28.7.3) and the VM exit occurred before those exceptions were either delivered or lost. In other cases, bit 12 is cleared to 0.
  - The setting of bit 14 (BS) is implementation-specific. However, it is not set if RFLAGS.TF = 0 or IA32_DEBUGCTL.BTF = 1.
- The reserved bits in the field are cleared.

- If the "save VMX-preemption timer value" VM-exit control is 1, the value of timer is saved into the VMX-preemption timer-value field. This is the value loaded from this field on VM entry as subsequently decremented (see Section 27.5.1). VM exits due to timer expiration save the value 0. Other VM exits may also save the value 0 if the timer expired during VM exit. (If the "save VMX-preemption timer value" VM-exit control is 0, VM exit does not modify the value of the VMX-preemption timer-value field.)
- If the logical processor supports the 1-setting of the "enable EPT" VM-execution control, values are saved into the four (4) PDPTE fields as follows:
  - If the "enable EPT" VM-execution control is 1 and the logical processor was using PAE paging at the time of the VM exit, the PDPTE values currently in use are saved:[1]
    - The values saved into bits 11:9 of each of the fields is undefined.

- If the value saved into one of the fields has bit 0 (present) clear, the value saved into bits 63:1 of that field is undefined. That value need not correspond to the value that was loaded by VM entry or to any value that might have been loaded in VMX non-root operation.
- If the value saved into one of the fields has bit 0 (present) set, the value saved into bits 63:12 of the field is a guest-physical address.

— If the "enable EPT" VM-execution control is 0 or the logical processor was not using PAE paging at the time of the VM exit, the values saved are undefined.

29.4 SAVING MSRS

After processor state is saved to the guest-state area, values of MSRs may be stored into the VM-exit MSR-store area (see Section 26.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-store count) is processed in order by storing the value of the MSR indexed by bits 31:0 (as they would be read by RDMSR) into bits 127:64. Processing of an entry fails in either of the following cases:

- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be read only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMBASE is an MSR that can be read only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be saved on VM exits for model-specific reasons. A processor may prevent certain MSRs (based on the value of bits 31:0) from being stored on VM exits, even if they can normally be read by RDMSR. Such model-specific behavior is documented in Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.
- Bits 63:32 of the entry are not all 0.
- An attempt to read the MSR indexed by bits 31:0 would cause a general-protection exception if executed via RDMSR with CPL = 0.

A VMX abort occurs if processing fails for any entry. See Section 29.7.

29.5 LOADING HOST STATE

Processor state is updated on VM exits in the following ways:

- Some state is loaded from or otherwise determined by the contents of the host-state area.
- Some state is determined by VM-exit controls.
- Some state is established in the same way on every VM exit.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order.

On processors that support Intel 64 architecture, the full values of each 64-bit field loaded (for example, the base address for GDTR) is loaded regardless of the mode of the logical processor before and after the VM exit.

The loading of host state is detailed in Section 29.5.1 to Section 29.5.5. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

A logical processor is in IA-32e mode after a VM exit only if the "host address-space size" VM-exit control is 1. If the logical processor was in IA-32e mode before the VM exit and this control is 0, a VMX abort occurs. See Section 29.7.

In addition to loading host state, VM exits clear address-range monitoring (Section 29.5.6).

---

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 5.4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM exit functions as if the "enable EPT" VM-execution control were 0. See Section 26.6.2.

After the state loading described in this section, VM exits may load MSRs from the VM-exit MSR-load area (see Section 29.6). This loading occurs only after the state loading described in this section.

29.5.1 Loading Host Control Registers, Debug Registers, MSRs

VM exits load new values for controls registers, debug registers, and some MSRs:

- CR0, CR3, and CR4 are loaded from the CR0 field, the CR3 field, and the CR4 field, respectively, with the following exceptions:
  - The following bits are not modified:
    - For CR0, ET, CD, NW; bits 63:32 (on processors that support Intel 64 architecture), 28:19, 17, and 15:6; and any bits that are fixed in VMX operation (see Section 25.8).[1]
    - For CR3, bits 63:52 and bits in the range 51:32 beyond the processor's physical-address width (they are cleared to 0).[2] (This item applies only to processors that support Intel 64 architecture.)
    - For CR4, any bits that are fixed in VMX operation (see Section 25.8).
  - CR4.PAE is set to 1 if the "host address-space size" VM-exit control is 1.
  - CR4.PCIDE is set to 0 if the "host address-space size" VM-exit control is 0.
- DR7 is set to 400H.
- If the "clear UINV" VM-exit control is 1, VM exit clears UINV.
- The following MSRs are established as follows:
  - The IA32_DEBUGCTL MSR is cleared to 00000000_00000000H.
  - The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since that field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
  - The IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are loaded from the IA32_SYSENTER_ESP and IA32_SYSENTER_EIP fields, respectively.

    If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.

    If the processor supports the Intel 64 architecture with $N < 64$ linear-address bits, each of bits 63:N is set to the value of bit N–1.[3]
  - The following steps are performed on processors that support Intel 64 architecture:
    - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 29.5.2).
    - The LMA and LME bits in the IA32_EFER MSR are each loaded with the setting of the "host address-space size" VM-exit control.
  - If the "load IA32_PERF_GLOBAL_CTRL" VM-exit control is 1, the IA32_PERF_GLOBAL_CTRL MSR is loaded from the IA32_PERF_GLOBAL_CTRL field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the "load IA32_PAT" VM-exit control is 1, the IA32_PAT MSR is loaded from the IA32_PAT field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the "load IA32_EFER" VM-exit control is 1, the IA32_EFER MSR is loaded from the IA32_EFER field. Bits that are reserved in that MSR are maintained with their reserved values.

1. Bits 28:19, 17, and 15:6 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. CR0.ET is always 1 and the other bits are always 0.
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

- If the "clear IA32_BNDCFGS" VM-exit control is 1, the IA32_BNDCFGS MSR is cleared to 00000000_00000000H; otherwise, it is not modified.
- If the "clear IA32_RTIT_CTL" VM-exit control is 1, the IA32_RTIT_CTL MSR is cleared to 00000000_00000000H; otherwise, it is not modified.
- If the "load CET" VM-exit control is 1, the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are loaded from the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR fields, respectively.

  If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.

  If the processor supports the Intel 64 architecture with N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.
- If the "load PKRS" VM-exit control is 1, the IA32_PKRS MSR is loaded from the IA32_PKRS field. Bits 63:32 of that MSR are maintained with zeroes.

With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-exit MSR-load area. See Section 29.6.

29.5.2 Loading Host Segment and Descriptor-Table Registers

Each of the registers CS, SS, DS, ES, FS, GS, and TR is loaded as follows (see below for the treatment of LDTR):

- The selector is loaded from the selector field. The segment is unusable if its selector is loaded with zero. The checks specified in Section 28.2.3 limit the selector values that may be loaded. In particular, CS and TR are never loaded with zero and are thus never unusable. SS can be loaded with zero only on processors that support Intel 64 architecture and only if the VM exit is to 64-bit mode (64-bit mode allows use of segments marked unusable).
- The base address is set as follows:
  - CS. Cleared to zero.
  - SS, DS, and ES. Undefined if the segment is unusable; otherwise, cleared to zero.
  - FS and GS. Undefined (but, on processors that support Intel 64 architecture, canonical) if the segment is unusable and the VM exit is not to 64-bit mode; otherwise, loaded from the base-address field.

    If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.[1] The values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
  - TR. Loaded from the host-state area. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.
- The segment limit is set as follows:
  - CS. Set to FFFFFFFFH (corresponding to a descriptor limit of FFFFFH and a G-bit setting of 1).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to FFFFFFFFH.
  - TR. Set to 00000067H.
- The type field and S bit are set as follows:
  - CS. Type set to 11 and S set to 1 (execute/read, accessed, non-conforming code segment).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, type set to 3 and S set to 1 (read/write, accessed, expand-up data segment).
  - TR. Type set to 11 and S set to 0 (busy 32-bit task-state segment).
- The DPL is set as follows:
  - CS, SS, and TR. Set to 0. The current privilege level (CPL) will be 0 after the VM exit completes.

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

   - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 0.
- The P bit is set as follows:
   - CS, TR. Set to 1.
   - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
- On processors that support Intel 64 architecture, CS.L is loaded with the setting of the "host address-space size" VM-exit control. Because the value of this control is also loaded into IA32_EFER.LMA (see Section 29.5.1), no VM exit is ever to compatibility mode (which requires IA32_EFER.LMA = 1 and CS.L = 0).
- D/B.
   - CS. Loaded with the inverse of the setting of the "host address-space size" VM-exit control. For example, if that control is 0, indicating a 32-bit guest, CS.D/B is set to 1.
   - SS. Set to 1.
   - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
   - TR. Set to 0.
- G.
   - CS. Set to 1.
   - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
   - TR. Set to 0.

The host-state area does not contain a selector field for LDTR. LDTR is established as follows on all VM exits: the selector is cleared to 0000H, the segment is marked unusable and is otherwise undefined.

The base addresses for GDTR and IDTR are loaded from the GDTR base-address field and the IDTR base-address field, respectively. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N of each base address is set to the value of bit N–1 of that base address. The GDTR and IDTR limits are each set to FFFFH.

29.5.3 Loading Host RIP, RSP, RFLAGS, and SSP

RIP and RSP are loaded from the RIP field and the RSP field, respectively. RFLAGS is cleared, except bit 1, which is always set.

If the "load CET" VM-exit control is 1, SSP (shadow-stack pointer) is loaded from the SSP field.

29.5.4 Checking and Loading Host Page-Directory-Pointer-Table Entries

If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LMA = 0, the logical processor uses **PAE paging**. See Section 5.4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.[1] When in PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs and, if they are valid, loads them into the processor (into internal, non-architectural registers).

A VM exit is to a VMM that uses PAE paging if (1) bit 5 (corresponding to CR4.PAE) is set in the CR4 field in the host-state area of the VMCS; and (2) the "host address-space size" VM-exit control is 0. Such a VM exit may check the validity of the PDPTEs referenced by the CR3 field in the host-state area of the VMCS. Such a VM exit must check their validity if either (1) PAE paging was not in use before the VM exit; or (2) the value of CR3 is changing as a result of the VM exit. A VM exit to a VMM that does not use PAE paging must not check the validity of the PDPTEs.

A VM exit that checks the validity of the PDPTEs uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use. If MOV to CR3 would cause a general-protection exception due to the

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

PDPTEs that would be loaded (e.g., because a reserved bit is set), a VMX abort occurs (see Section 29.7). If a VM exit to a VMM that uses PAE does not cause a VMX abort, the PDPTEs are loaded into the processor as would MOV to CR3, using the value of CR3 being load by the VM exit.

29.5.5 Updating Non-Register State

VM exits affect the non-register state of a logical processor as follows:

- A logical processor is always in the active state after a VM exit.
- Event blocking is affected as follows:
  - — There is no blocking by STI or by MOV SS after a VM exit.
  - — VM exits caused directly by non-maskable interrupts (NMIs) cause blocking by NMI (see Table 26-3). Other VM exits do not affect blocking by NMI. (See Section 29.1 for the case in which an NMI causes a VM exit indirectly.)
- There are no pending debug exceptions after a VM exit.

Section 30.4 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM exits invalidate cached mappings:

- If the “enable VPID” VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EPTRTA values (EPTRTA is the value of bits 51:12 of EPTP).
- VM exits are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the “enable VPID” VM-execution control is 1.

29.5.6 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 10.10.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. VM exits clear any address-range monitoring that may be in effect.

29.6 LOADING MSRS

VM exits may load MSRs from the VM-exit MSR-load area (see Section 26.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C0000100H (the IA32_FS_BASE MSR) or C0000101H (the IA32_GS_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMM_MONITOR_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM exits for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.
- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.[1]

If processing fails for any entry, a VMX abort occurs. See Section 29.7.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM exit, the logical processor does not use any translations that were cached before the transition.

29.7 VMX ABORTS

A problem encountered during a VM exit leads to a **VMX abort**. A VMX abort takes a logical processor into a shutdown state as described below.

A VMX abort does not modify the VMCS data in the VMCS region of any active VMCS. The contents of these data are thus suspect after the VMX abort.

On a VMX abort, a logical processor saves a nonzero 32-bit VMX-abort indicator field at byte offset 4 in the VMCS region of the VMCS whose misconfiguration caused the failure (see Section 26.2). The following values are used:

1. There was a failure in saving guest MSRs (see Section 29.4).
2. Host checking of the page-directory-pointer-table entries (PDPTEs) failed (see Section 29.5.4).
3. The current VMCS has been corrupted (through writes to the corresponding VMCS region) in such a way that the logical processor cannot complete the VM exit properly.
4. There was a failure on loading host MSRs (see Section 29.6).
5. There was a machine-check event during VM exit (see Section 29.8).
6. The logical processor was in IA-32e mode before the VM exit and the "host address-space size" VM-exit control was 0 (see Section 29.5).

Some of these causes correspond to failures during the loading of state from the host-state area. Because the loading of such state may be done in any order (see Section 29.5) a VM exit that might lead to a VMX abort for multiple reasons (for example, the current VMCS may be corrupt and the host PDPTEs might not be properly configured). In such cases, the VMX-abort indicator could correspond to any one of those reasons.

A logical processor never reads the VMX-abort indicator in a VMCS region and writes it only with one of the non-zero values mentioned above. The VMX-abort indicator allows software on one logical processor to diagnose the VMX-abort on another. For this reason, it is recommended that software running in VMX root operation zero the VMX-abort indicator in the VMCS region of any VMCS that it uses.

After saving the VMX-abort indicator, operation of a logical processor experiencing a VMX abort depends on whether the logical processor is in SMX operation:[1]

- If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000DH, indicating "VMX abort." See the Intel® Trusted Execution Technology Measured Launched Environment Programming Guide.
- If the logical processor is outside SMX operation, it issues a special bus cycle (to notify the chipset) and enters the **VMX-abort shutdown state**. RESET is the only event that wakes a logical processor from the VMX-abort shutdown state. The following events do not affect a logical processor in this state: machine-check events; INIT signals; external interrupts; non-maskable interrupts (NMIs); start-up IPIs (SIPIs); and system-management interrupts (SMIs).

29.8 MACHINE-CHECK EVENTS DURING VM EXIT

If a machine-check event occurs during VM exit, one of the following occurs:

1. Note the following about processors that support Intel 64 architecture. If CR0.PG = 1, WRMSR to the IA32_EFER MSR causes a general-protection exception if it would modify the LME bit. Since CR0.PG is always 1 in VMX operation, the IA32_EFER MSR should not be included in the VM-exit MSR-load area for the purpose of modifying the LME bit.

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 7, "Safer Mode Extensions Reference," in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

- The machine-check event is handled as if it occurred before the VM exit:
  - If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:[1]
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000CH, indicating "unrecoverable machine-check condition."
    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If CR4.MCE = 1, a machine-check exception (#MC) is generated:
    - If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
    - If bit 18 of the exception bitmap is 1, the exception causes a VM exit.
- The machine-check event is handled after VM exit completes:
  - If the VM exit ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).
    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If the VM exit ends with CR4.MCE = 1, a machine-check exception (#MC) is delivered through the host IDT.
- A VMX abort is generated (see Section 29.7). The logical processor blocks events as done normally in VMX abort. The VMX abort indicator is 5, for "machine-check event during VM exit."

The first option is not used if the machine-check event occurs after any host state has been loaded. The second option is used only if VM entry is able to load all host state.

29.9 USER-INTERRUPT RECOGNITION AFTER VM EXIT

A VM exit results in recognition of a pending user interrupt if it completes with CR4.UINTR = IA32_EFER.LMA = 1 and with UIRR ≠ 0; otherwise, no pending user interrupt is recognized.

15. Updates to Chapter 39, Volume 3D

Change bars and violet text show changes to Chapter 39 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter:

- Added EUPDATESVN to
  - Table 39-1, “Register Usage of Privileged Enclave Instruction Leaf Functions” in Section 39.1.1, “ENCLS Register Usage Summary.”
  - Table 39-4, “Error or Information Codes for Intel® SGX Instructions” in Section 39.1.4, “Information and Error Codes.”
  - Table 39-7, “Additional Concurrency Restrictions” in Section 39.1.6.1, “Concurrency Tables of Intel® SGX Instructions.”
- Added content for ENCLS EUPDATESVN instruction in Section 39.3, “Intel® SGX System Leaf Function Reference.”

CHAPTER 39 INTEL® SGX INSTRUCTION REFERENCES

This chapter describes the supervisor and user level instructions provided by Intel® Software Guard Extensions (Intel® SGX). In general, various functionality is encoded as leaf functions within the ENCLS (supervisor), ENCLU (user), and the ENCLV (virtualization operation) instruction mnemonics. Different leaf functions are encoded by specifying an input value in the EAX register of the respective instruction mnemonic.

39.1 INTEL® SGX INSTRUCTION SYNTAX AND OPERATION

ENCLS, ENCLU, and ENCLV instruction mnemonics for all leaf functions are covered in this section.

For all instructions, the value of CS.D is ignored; addresses and operands are 64 bits in 64-bit mode and are otherwise 32 bits. Aside from EAX specifying the leaf number as input, each instruction leaf may require all or some subset of the RBX/RCX/RDX as input parameters. Some leaf functions may return data or status information in one or more of the general purpose registers.

39.1.1 ENCLS Register Usage Summary

Table 39-1 summarizes the implicit register usage of supervisor mode enclave instructions.

Table 39-1. Register Usage of Privileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
ECREATE	00H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EADD	01H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EINIT	02H (In)	SIGSTRUCT (In, EA)	SECS (In, EA)	EINITTOKEN (In, EA)
EREMOVE	03H (In)		EPCPAGE (In, EA)	
EDBGRD	04H (In)	Result Data (Out)	EPCPAGE (In, EA)	
EDBGWR	05H (In)	Source Data (In)	EPCPAGE (In, EA)	
EEXTEND	06H (In)	SECS (In, EA)	EPCPAGE (In, EA)	
ELDB	07H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDU	08H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
EBLOCK	09H (In)		EPCPAGE (In, EA)	
EPA	0AH (In)	PT_VA (In)	EPCPAGE (In, EA)	
EWB	0BH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ETRACK	0CH (In)		EPCPAGE (In, EA)	
EAUG	0DH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EMODPR	0EH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODT	0FH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
ERDINFO	010H (In)	RDINFO (In, EA*)	EPCPAGE (In, EA)	
ETRACKC	011H (In)		EPCPAGE (In, EA)	
ELDBC	012H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDUC	013H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)
EUPDATESVN	018H			
EA: Effective Address				

39.1.2 ENCLU Register Usage Summary

Table 39-2 summarizes the implicit register usage of user mode enclave instructions.

Table 39-2. Register Usage of Unprivileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
EREPORT	00H (In)	TARGETINFO (In, EA)	REPORTDATA (In, EA)	OUTPUTDATA (In, EA)
EGETKEY	01H (In)	KEYREQUEST (In, EA)	KEY (In, EA)	
EENTER	02H (In)	TCS (In, EA)	AEP (In, EA)	
	RBX.CSSA (Out)		Return (Out, EA)	
ERESUME	03H (In)	TCS (In, EA)	AEP (In, EA)	
EEXIT	04H (In)	Target (In, EA)	Current AEP (Out)	
EACCEPT	05H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODPE	06H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EACCEPTCOPY	07H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	EPCPAGE (In, EA)
EDECCSSA	09H (In)			
EA: Effective Address				

39.1.3 ENCLV Register Usage Summary

Table 39-3 summarizes the implicit register usage of virtualization operation enclave instructions.

Table 39-3. Register Usage of Virtualization Operation Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
EDECVIRTCHILD	00H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
EINCVIRTCHILD	01H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
ESETCONTEXT	02H (In)		EPCPAGE (In, EA)	Context Value (In, EA)
EA: Effective Address				

39.1.4 Information and Error Codes

Information and error codes are reported by various instruction leaf functions to show an abnormal termination of the instruction or provide information which may be useful to the developer. Table 39-4 shows the various codes and the instruction which generated the code. Details of the meaning of the code is provided in the individual instruction.

Table 39-4. Error or Information Codes for Intel® SGX Instructions

Name	Value	Returned By
No Error	0	
SGX_INVALID_SIG_STRUCT	1	EINIT
SGX_INVALID_ATTRIBUTE	2	EINIT, EGETKEY
SGX_BLKSTATE	3	EBLOCK
SGX_INVALID_MEASUREMENT	4	EINIT
SGX_NOTBLOCKABLE	5	EBLOCK
SGX_PG_INVLD	6	EBLOCK, ERDINFO, ETRACKC

Table 39-4. Error or Information Codes for Intel® SGX Instructions

Name	Value	Returned By
SGX_EPC_PAGE_CONFLICT	7	EBLOCK, EMODPR, EMODT, ERDINFO , EDECVIRTCHILD, EINCVIRTCHILD, ELDBC, ELDUC, ESETCONTEXT, ETRACKC, EUPDATESVN
SGX_INVALID_SIGNATURE	8	EINIT
SGX_MAC_COMPARE_FAIL	9	ELDB, ELDU, ELDBC, ELDUC
SGX_PAGE_NOT_BLOCKED	10	EWB
SGX_NOT_TRACKED	11	EWB, EACCEPT
SGX_VA_SLOT_OCCUPIED	12	EWB
SGX_CHILD_PRESENT	13	EWB, EREMOVE
SGX_ENCLAVE_ACT	14	EREMOVE
SGX_ENTRYEPOCH_LOCKED	15	EBLOCK
SGX_INVALID_EINITTOKEN	16	EINIT
SGX_PREV_TRK_INCMPL	17	ETRACK, ETRACKC
SGX_PG_IS_SECS	18	EBLOCK
SGX_PAGE_ATTRIBUTES_MISMATCH	19	EACCEPT, EACCEPTCOPY
SGX_PAGE_NOT_MODIFIABLE	20	EMODPR, EMODT
SGX_PAGE_NOT_DEBUGGABLE	21	EDBGRD, EDBGWR
SGX_INVALID_COUNTER	25	EDECVIRTCHILD
SGX_PG_NONEPC	26	ERDINFO
SGX_TRACK_NOT_REQUIRED	27	ETRACKC
SGX_INSUFFICIENT_ENTROPY	29	EUPDATESVN
SGX_EPC_NOT_READY	30	EUPDATESVN
SGX_NO_UPDATE	31	EUPDATESVN
SGX_INVALID_CPUSVN	32	EINIT, EGETKEY
SGX_INVALID_ISVSVN	64	EGETKEY
SGX_UNMASKED_EVENT	128	EINIT
SGX_INVALID_KEYNAME	256	EGETKEY

39.1.5 Internal CREGs

The CREGs as shown in Table 5-4 are hardware specific registers used in this document to indicate values kept by the processor. These values are used while executing in enclave mode or while executing an Intel SGX instruction. These registers are not software visible and are implementation specific. The values in Table 39-5 appear at various places in the pseudo-code of this document. They are used to enhance understanding of the operations.

Table 39-5. List of Internal CREG

Name	Size (Bits)	Scope
CR_ENCLAVE_MODE	1	LP
CR_DBGOPTIN	1	LP
CR_TCS_LA	64	LP
CR_TCS_PA	64	LP
CR_ACTIVE_SECS	64	LP
CR_ELRANGE	128	LP
CR_SAVE_TF	1	LP

Table 39-5. List of Internal CREG

Name	Size (Bits)	Scope
CR_SAVE_FS	64	LP
CR_GPR_PA	64	LP
CR_XSAVE_PAGE_n	64	LP
CR_SAVE_DR7	64	LP
CR_SAVE_PERF_GLOBAL_CTRL	64	LP
CR_SAVE_DEBUGCTL	64	LP
CR_SAVE_PEBS_ENABLE	64	LP
CR_CPUSVN	128	PACKAGE
CR_SGXOWNEREPOCH	128	PACKAGE
CR_SAVE_XCR0	64	LP
CR_SGX_ATTRIBUTES_MASK	128	LP
CR_PAGING_VERSION	64	PACKAGE
CR_VERSION_THRESHOLD	64	PACKAGE
CR_NEXT_EID	64	PACKAGE
CR_BASE_PK	128	PACKAGE
CR_SEAL_FUSES	128	PACKAGE
CR_CET_SAVE_AREA_PA	64	LP
CR_ENCLAVE_SS_TOKEN_PA	64	LP
CR_SAVE_IA32_U_CET	64	LP
CR_SAVE_SSP	64	LP

39.1.6 Concurrent Operation Restrictions

Under certain conditions, Intel SGX disallows certain leaf functions from operating concurrently. Listed below are some examples of concurrency that are not allowed.

- For example, Intel SGX disallows the following leafs to concurrently operate on the same EPC page.
  - ECREATE, EADD, and EREMOVE are not allowed to operate on the same EPC page concurrently with themselves.
  - EADD, EEXTEND, and EINIT leaves are not allowed to operate on the same SECS concurrently.
- Intel SGX disallows the EREMOVE leaf from removing pages from an enclave that is in use.
- Intel SGX disallows entry (EENTER and ERESUME) to an enclave while a page from that enclave is being removed.

When disallowed operation is detected, a leaf function may do one of the following:

- Return an SGX_EPC_PAGE_CONFLICT error code in RAX.
- Cause a #GP(0) exception.

To prevent such exceptions, software must serialize leaf functions or prevent these leaf functions from accessing the same EPC page.

39.1.6.1 Concurrency Tables of Intel® SGX Instructions

The tables below detail the concurrent operation restrictions of all SGX leaf functions. For each leaf function, the table has a separate line for each of the EPC pages the leaf function accesses.

For each such EPC page, the base concurrency requirements are detailed as follows:

- **Exclusive Access** means that no other leaf function that requires either shared or exclusive access to the same EPC page may be executed concurrently. For example, EADD requires an exclusive access to the target page it accesses.
- **Shared Access** means that no other leaf function that requires an exclusive access to the same EPC page may be executed concurrently. Other leaf functions that require shared access may run concurrently. For example, EADD requires a shared access to the SECS page it accesses.
- **Concurrent Access** means that any other leaf function that requires any access to the same EPC page may be executed concurrently. For example, EGETKEY has no concurrency requirements for the KEYREQUEST page.

In addition to the base concurrency requirements, additional concurrency requirements are listed, which apply only to specific sets of leaf functions. For example, there are additional requirements that apply for EADD, EXTEND, and EINIT. EADD and EEXTEND can't execute concurrently on the same SECS page.

The tables also detail the leaf function's behavior when a conflict happens, i.e., a concurrency requirement is not met. In this case, the leaf function may return an SGX_EPC_PAGE_CONFLICT error code in RAX, or it may cause an exception. In addition, the tables detail those conflicts where a VM Exit may be triggered, and list the Exit Qualification code that is provided in such cases.

Table 39-6. Base Concurrency Restrictions

Leaf	Parameter		Base Concurrency Restrictions		
			Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target	[DS:RCX]	Shared	#GP	
	SECINFO	[DS:RBX]	Concurrent		
EACCEPTCOPY	Target	[DS:RCX]	Concurrent		
	Source	[DS:RDX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EADD	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EAUG	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EBLOCK	Target	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
ECREATE	SECS	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EDBGRD	Target	[DS:RCX]	Shared	#GP	
EDBGWR	Target	[DS:RCX]	Shared	#GP	
EDECVIRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RCX]	Concurrent		
EENTERTCS	SECS	[DS:RBX]	Shared	#GP	
EEXIT			Concurrent		
EEXTEND	Target	[DS:RCX]	Shared	#GP	
	SECS	[DS:RBX]	Concurrent		
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		
	OUTPUTDATA	[DS:RCX]	Concurrent		
EINCVIRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RCX]	Concurrent		

Table 39-6. Base Concurrency Restrictions

Leaf	Parameter		Base Concurrency Restrictions		
			Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINIT	SECS	[DS:RCX]	Shared	#GP	
ELDB/ELDU	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EDLBC/ELDUC	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE _CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA	[DS:RDX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	SGX_EPC_PAGE _CONFLICT	
EMODPE	Target	[DS:RCX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EMODPR	Target	[DS:RCX]	Shared	#GP	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE _CONFLICT	EPC_PAGE_CONFLICT_ERROR
EPA	VA	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
ERDINFO	Target	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
EREMOVE	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		
	REPORTDATA	[DS:RCX]	Concurrent		
	OUTPUTDATA	[DS:RDX]	Concurrent		
ERESUME	TCS	[DS:RBX]	Shared	#GP	
ESETCONTEXT	SECS	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
ETRACK	SECS	[DS:RCX]	Shared	#GP	
ETRACKC	Target	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	Implicit	Concurrent		
EWB	Source	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	

Table 39-7. Additional Concurrency Restrictions

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EACCEPTCOPY	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EADD	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Exclusive	#GP	Concurrent	
EAUG	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EBLOCK	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ECREATE	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGRD	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGWR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDECVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EENTERTCS	SECS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EEXIT			Concurrent		Concurrent		Concurrent	
EEXTEND	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINCVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINIT	SECS	[DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	
ELDB/ELDU	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EDLBC/ELDUC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EMODPE	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	

Table 39-7. Additional Concurrency Restrictions

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPR	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	
EPA	VA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ERDINFO	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREMOVE	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
ERESUME	TCS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
ESETCONTEXT	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ETRACK	SECS	[DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT[1]
ETRACKC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	Implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT[1]
EUPDATESVN	EPCM		Exclusive	SGX_EPC_PAGE_CONFLICT	Exclusive	SGX_EPC_PAGE_CONFLICT	Exclusive	SGX_EPC_PAGE_CONFLICT
EWB	Source	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	

NOTES:

1. SGX_CONFLICT VM Exit Qualification =TRACKING_RESOURCE_CONFLICT.

39.2 INTEL® SGX INSTRUCTION REFERENCE

ENCLS—Execute an Enclave System Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CF ENCLS	ZO	V/V	NA	This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
ZO	NA	NA	NA	See Section 39.3

Description

The ENCLS instruction invokes the specified privileged Intel SGX leaf function for managing and debugging enclaves. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLS instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

In VMX non-root operation, execution of ENCLS may cause a VM exit if the "enable ENCLS exiting" VM-execution control is 1. In this case, execution of individual leaf functions of ENCLS is governed by the ENCLS-exiting bitmap field in the VMCS. Each bit in that field corresponds to the index of an ENCLS leaf function (as provided in EAX).

Software in VMX root operation can thus intercept the invocation of various ENCLS leaf functions in VMX non-root operation by setting the "enable ENCLS exiting" VM-execution control and setting the corresponding bits in the ENCLS-exiting bitmap.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 || CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

Operation

```
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation and the "enable ENCLS exiting" VM-execution control is 1
    THEN
         IF EAX < 63 and ENCLS_exiting_bitmap[EAX] = 1 or EAX> 62 and ENCLS_exiting_bitmap[63] = 1
              THEN VM exit;
         FI;
FI;
IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF (EAX is an invalid leaf number)
    THEN #GP(0); FI;
```

```
IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/66H/REP/VEX prefixes are used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If data segment expand down.
	If CR0.PG=0.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/66H/REP/VEX prefixes are used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.

ENCLU—Execute an Enclave User Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D7 ENCLU	ZO	V/V	NA	This instruction is used to execute non-privileged Intel SGX leaf functions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
ZO	NA	NA	NA	See Section 39.4

Description

The ENCLU instruction invokes the specified non-privileged Intel SGX leaf functions. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLU instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute this instruction when CPL < 3 results in #UD. The instruction produces a general-protection exception (#GP) if either CR0.PG or CR0.NE is 0, or if an attempt is made to invoke an undefined leaf function. The ENCLU instruction produces a device not available exception (#NM) if CR0.TS = 1.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 or CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 and CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

Operation

```
IN_64BIT_MODE := 0;
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

(* If enclosing app has CET indirect branch tracking enabled then if it is not ERESUME leaf cause a #CP fault *)
(* If the ERESUME is not successful it will leave tracker in WAIT_FOR_ENDBRANCH *)
TRACKER = (CPL == 3) ? IA32_U_CET.TRACKER : IA32_S_CET.TRACKER
IF EndbranchEnabledAndNotSuppressed(CPL) and TRACKER = WAIT_FOR_ENDBRANCH and
  (EAX != ERESUME or CR0.TS or (in SMM) or (CPUID.SGX_LEAF.0:EAX.SE1 = 0) or (CPL < 3))
    THEN
        Handle CET State machine violation         (* see Section 18.3.6, "Legacy Compatibility Treatment," in the
                                                    Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. *)
FI;

IF CR0.PE= 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF CR0.TS = 1
    THEN #NM; FI;

IF CPL < 3
    THEN #UD; FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;
```

```
IF EAX is invalid leaf number
    THEN #GP(0); FI;

IF CR0.PG = 0 or CR0.NE = 0
    THEN #GP(0); FI;

IN_64BIT_MODE := IA32_EFER.LMA AND CS.L ? 1 : 0;
(* Check not in 16-bit mode and DS is not a 16-bit segment *)
IF not in 64-bit mode and CS.D = 0
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 1 and (EAX = 2 or EAX = 3) (* EENTER or ERESUME *)
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 0 and (EAX = 0 or EAX = 1 or EAX = 4 or EAX = 5 or EAX = 6 or EAX = 7 or EAX = 9)
(* EREPORT, EGETKEY, EEXIT, EACCEPT, EMODPE, EACCEPTCOPY, or EDECCSSA *)
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/66H/REP/VEX prefixes are used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If operating in 16-bit mode.
	If data segment is in 16-bit mode.
	If CR0.PG = 0 or CR0.NE= 0.
#NM	If CR0.TS = 1.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/66H/REP/VEX prefixes are used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If CR0.NE= 0.
#NM	If CR0.TS = 1.

ENCLV—Execute an Enclave VMM Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C0 ENCLV	ZO	V/V	NA	This instruction is used to execute privileged SGX leaf functions that are reserved for VMM use. They are used for managing the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
ZO	NA	NA	NA	See Section 39.3

Description

The ENCLV instruction invokes the virtualization SGX leaf functions for managing enclaves in a virtualized environment. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In non 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLV instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, if it is executed in system-management mode (SMM), or not in VMX operation. Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

Software in VMX root mode of operation can enable execution of the ENCLV instruction in VMX non-root mode by setting enable ENCLV execution control in the VMCS. If enable ENCLV execution control in the VMCS is clear, execution of the ENCLV instruction in VMX non-root mode results in #UD.

When execution of ENCLV instruction in VMX non-root mode is enabled, software in VMX root operation can intercept the invocation of various ENCLV leaf functions in VMX non-root operation by setting the corresponding bits in the ENCLV-exiting bitmap.

Addresses and operands are 32 bits in 32-bit mode (IA32_EFER.LMA == 0 || CS.L == 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA == 1 && CS.L == 1). CS.D value has no impact on address calculation.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

Operation

```
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.OSS = 0
    THEN #UD; FI;

IF not in VMX Operation or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation
  IF "enable ENCLV exiting" VM-execution control is 1
    THEN
      IF EAX < 63 and ENCLV_exiting_bitmap[EAX] = 1 or EAX> 62 and ENCLV_exiting_bitmap[63] = 1
          THEN VM exit;
      FI;
   ELSE
    #UD; FI;
```

```
FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF (EAX is an invalid leaf number)
    THEN #GP(0); FI;

IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions.

Protected Mode Exceptions

#UD	If any of the LOCK/66H/REP/VEX prefixes are used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If data segment expand down.
	If CR0.PG=0.

Real-Address Mode Exceptions

#UD	ENCLV is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLV is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/66H/REP/VEX prefixes are used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.

39.3 INTEL® SGX SYSTEM LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLS instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EADD—Add a Page to an Uninitialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLS[EADD]	IR	V/V	SGX1	This leaf function adds a page to an uninitialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EADD (In)	Address of a PAGEINFO (In)	Address of the destination EPC page (In)

Description

This leaf function copies a source page from non-enclave memory into the EPC, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in EPCM. As part of the association, the enclave offset and the security attributes are measured and extended into the SECS.MRENCLAVE. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of EADD leaf function.

EADD Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

EADD Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	If security attributes specifies a TCS and the source page specifies unsupported TCS values or fields.
The SECS has been initialized.	The specified enclave offset is outside of the enclave address space.

Concurrency Restrictions

Table 39-8. Base Concurrency Restrictions of EADD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EADD	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

Table 39-9. Additional Concurrency Restrictions of EADD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EADD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EADD Operational Flow

Name	Type	Size (bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECS := DS:RBX.SECS;
TMP_SECINFO := DS:RBX.SECINFO;
TMP_LINADDR := DS:RBX.LINADDR;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECS is not 4KByte aligned or
    DS:TMP_SECINFO is not 64Byte aligned or TMP_LINADDR is not 4KByte aligned)
    THEN #GP(0); FI;

IF (DS:TMP_SECS does not resolve within an EPC)
    THEN #PF(DS:TMP_SECS); FI;

SCRATCH_SECINFO := DS:TMP_SECINFO;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero or
    ! (SCRATCH_SECINFO.FLAGS.PT is PT_REG or SCRATCH_SECINFO.FLAGS.PT is PT_TCS or
    (SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
    (SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1)) )
    THEN #GP(0); FI;

(* If PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST OR
    SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST) AND CR4.CET == 0)
    THEN #GP(0); FI;

(* Check the EPC page for concurrency *)
IF (EPC page is not available for EADD)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address := DS:RCX;
            Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EADD)
    THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
    THEN #PF(DS:TMP_SECS); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPGE[32767:0];

CASE (SCRATCH_SECINFO.FLAGS.PT)

    PT_TCS:
        IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
        IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and
            ((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH) )) #GP(0); FI;
        (* Ensure TCS.PREVSSP is zero *)
        IF (CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1) and (DS:RCX.PREVSSP != 0) #GP(0); FI;
        BREAK;
    PT_REG:
        IF (SCRATCH_SECINFO.FLAGS.W = 1 and SCRATCH_SECINFO.FLAGS.R = 0) #GP(0); FI;
        BREAK;
    PT_SS_FIRST:
    PT_SS_REST:
    (* SS pages cannot be created on first or last page of ELRANGE *)
    IF ( TMP_LINADDR = DS:TMP_SECS.BASEADDR or TMP_LINADDR = (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
        THEN #GP(0); FI;
    IF ( DS:RCX[4087:0] != 0 ) #GP(0); FI;
    IF (SCRATCH_SECINFO.FLAGS.PT == PT_SS_FIRST)
        THEN
            (* Check that valid RSTORSSP token exists *)
            IF ( DS:RCX[4095:4088] != ((TMP_LINADDR + 0x1000) | DS:TMP_SECS.ATTRIBUTES.MODE64BIT) ) #GP(0); FI;
        ELSE
            (* Check the 8 bytes are zero *)
            IF ( DS:RCX[4095:4088] != 0 ) #GP(0); FI;
    FI;
    IF (SCRATCH_SECINFO.FLAGS.W = 0 OR SCRATCH_SECINFO.FLAGS.R = 0 OR
     SCRATCH_SECINFO.FLAGS.X = 1) #GP(0); FI;
        BREAK;
ESAC;

(* Check the enclave offset is within the enclave linear address space *)
IF (TMP_LINADDR < DS:TMP_SECS.BASEADDR or TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE)
    THEN #GP(0); FI;

(* Check concurrency of measurement resource*)
IF (Measurement being updated)
    THEN #GP(0); FI;

(* Check if the enclave to which the page will be added is already in Initialized state *)
IF (DS:TMP_SECS already initialized)
    THEN #GP(0); FI;

(* For TCS pages, force EPCM.rwx bits to 0 and no debug access *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
    THEN
        SCRATCH_SECINFO.FLAGS.R := 0;
        SCRATCH_SECINFO.FLAGS.W := 0;
        SCRATCH_SECINFO.FLAGS.X := 0;
        (DS:RCX).FLAGS.DBGOPTIN := 0; // force TCS.FLAGS.DBGOPTIN off
        DS:RCX.CSSA := 0;
        DS:RCX.AEP := 0;
        DS:RCX.STATE := 0;
FI;

(* Add enclave offset and security attributes to MRENCLAVE *)
TMP_ENCLAVEOFFSET := TMP_LINADDR - DS:TMP_SECS.BASEADDR;
TMPUPDATEFIELD[63:0] := 0000000044444145H; // "EADD"
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] := SCRATCH_SECINFO[375:0]; // 48 bytes
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(* Add enclave offset and security attributes to MRENCLAVE *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;
```

```
(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).VALID := 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If an enclave memory operand is outside of the EPC.
	If an enclave memory operand is the wrong type.
	If a memory operand is locked.
	If the enclave is initialized.
	If the enclave's MRENCLAVE is locked.
	If the TCS page reserved bits are set.
	If the TCS page PREVSSP field is not zero.
	If the PT_SS_REST or PT_SS_REST page is the first or last page in the enclave.
	If the PT_SS_FIRST or PT_SS_REST page is not initialized correctly.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If an enclave memory operand is outside of the EPC.
	If an enclave memory operand is the wrong type.
	If a memory operand is locked.
	If the enclave is initialized.
	If the enclave's MRENCLAVE is locked.
	If the TCS page reserved bits are set.
	If the TCS page PREVSSP field is not zero.
	If the PT_SS_REST or PT_SS_REST page is the first or last page in the enclave.
	If the PT_SS_FIRST or PT_SS_REST page is not initialized correctly.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the EPC page is valid.

EAUG—Add a Page to an Initialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0DH ENCLS[EAUG]	IR	V/V	SGX2	This leaf function adds a page to an initialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EAUG (In)	Address of a PAGEINFO (In)	Address of the destination EPC page (In)

Description

This leaf function zeroes a page of EPC memory, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in the EPCM. As part of the association, the security attributes are configured to prevent access to the EPC page until a corresponding invocation of the EACCEPT leaf or EACCEPT-COPY leaf confirms the addition of the new page into the enclave. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EAUG leaf function.

EAUG Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Must be zero	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

EAUG Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	The specified enclave offset is outside of the enclave address space.
The SECS has been initialized.	

Concurrency Restrictions

Table 39-10. Base Concurrency Restrictions of EAUG

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EAUG	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

Table 39-11. Additional Concurrency Restrictions of EAUG

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EAUG	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EAUG Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SECS := DS:RBX.SECS;
TMP_SECINFO := DS:RBX.SECINFO;
IF (DS:RBX.SECINFO is not 0)
    THEN
        IF (DS:TMP_SECINFO is not 64B aligned)
            THEN #GP(0); FI;
FI;

TMP_LINADDR := DS:RBX.LINADDR;

IF ( DS:TMP_SECS is not 4KByte aligned or TMP_LINADDR is not 4KByte aligned )
    THEN #GP(0); FI;

IF DS:RBX.SRCPAGE is not 0
    THEN #GP(0); FI;

IF (DS:TMP_SECS does not resolve within an EPC)
    THEN #PF(DS:TMP_SECS); FI;

(* Check the EPC page for concurrency *)
```

```
IF (EPC page in use)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address := DS:RCX;
                Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI:

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

(* copy SECINFO contents into a scratch SECINFO *)
IF (DS:RBX.SECINFO is 0)
    THEN
        (* allocate and initialize a new scratch SECINFO structure *)
        SCRATCH_SECINFO.PT := PT_REG;
        SCRATCH_SECINFO.R := 1;
        SCRATCH_SECINFO.W := 1;
        SCRATCH_SECINFO.X := 0;
        << zero out remaining fields of SCRATCH_SECINFO >>
    ELSE
        (* copy SECINFO contents into scratch SECINFO *)
        SCRATCH_SECINFO := DS:TMP_SECINFO;
        (* check SECINFO flags for misconfiguration *)
        (* reserved flags must be zero *)
        (* SECINFO.FLAGS.PT must either be PT_SS_FIRST, or PT_SS_REST *)
        IF ( (SCRATCH_SECINFO reserved fields are not 0) or
        CPUID.(EAX=12H, ECX=1):EAX[6] is 0) OR
         (SCRATCH_SECINFO.PT is not PT_SS_FIRST, or PT_SS_REST) OR
         ( (SCRATCH_SECINFO.FLAGS.R is 0) OR (SCRATCH_SECINFO.FLAGS.W is 0) OR (SCRATCH_SECINFO.FLAGS.X is 1) ) )
            THEN #GP(0); FI;
FI;
(* Check if PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) AND CR4.CET == 0 )
    THEN #GP(0); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EAUG)
    THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
    THEN #PF(DS:TMP_SECS); FI;

(* Check if the enclave to which the page will be added is in the Initialized state *)
IF (DS:TMP_SECS is not initialized)
    THEN #GP(0); FI;
```

```
(* Check the enclave offset is within the enclave linear address space *)
IF ( (TMP_LINADDR < DS:TMP_SECS.BASEADDR) or (TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE) )
    THEN #GP(0); FI;

IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) )
    THEN
        (* SS pages cannot created on first or last page of ELRANGE *)
        IF ( TMP_LINADDR == DS:TMP_SECS.BASEADDR OR
         TMP_LINADDR == (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
            THEN
                #GP(0); FI;
FI;

(* Clear the content of EPC page*)
DS:RCX[32767:0] := 0;

IF (CPUID.(EAX=07H, ECX=0H):ECX[CET_SS] = 1)
    THEN
        (* set up shadow stack RSTORSSP token *)
        IF (SCRATCH_SECINFO.PT is PT_SS_FIRST)
        THEN
            DS:RCX[0xFF8] := (TMP_LINADDR + 0x1000) | TMP_SECS.ATTRIBUTES.MODE64BIT; FI;
FI;

(* Set EPCM security attributes *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 1;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;

(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM valid fields *)
EPCM(DS:RCX).VALID := 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
	If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
	If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

EBLOCK—Mark a page in EPC as Blocked

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 09H ENCLS[EBLOCK]	IR	V/V	SGX1	This leaf function marks a page in the EPC as blocked.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	EBLOCK (In)	Return error code (Out)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be marked as BLOCKED. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

An error code is returned in RAX.

The table below provides additional information on the memory parameter of EBLOCK leaf function.

EBLOCK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 39-12. EBLOCK Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EBLOCK successful.
SGX_BLKSTATE	Page already blocked. This value is used to indicate to a VMM that the page was already in BLOCKED state as a result of EBLOCK and thus will need to be restored to this state when it is eventually reloaded (using ELDB).
SGX_ENTRYEPOCH_LOCKED	SECS locked for Entry Epoch update. This value indicates that an ETRACK is currently executing on the SECS. The EBLOCK should be reattempted.
SGX_NOTBLOCKABLE	Page type is not one which can be blocked.
SGX_PG_INVLD	Page is not valid and cannot be blocked.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

Concurrency Restrictions

Table 39-13. Base Concurrency Restrictions of EBLOCK

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EBLOCK	Target [DS:RCX]	Shared	SGX_EPC_PAGE_ CONFLICT	

Table 39-14. Additional Concurrency Restrictions of EBLOCK

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EBLOCK	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EBLOCK Operational Flow

Name	Type	Size (Bits)	Description
TMP_BLKSTATE	Integer	64	Page is already blocked.

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;

(* Check the EPC page for concurrency*)
IF (EPC page in use)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PG_INVLD;
        GOTO DONE;
FI;

IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_TRIM)
 and EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
    THEN
        RFLAGS.CF := 1;
        IF (EPCM(DS:RCX).PT = PT_SECS)
            THEN RAX := SGX_PG_IS_SECS;
            ELSE RAX := SGX_NOTBLOCKABLE;
        FI;
        GOTO DONE;
FI;

(* Check if the page is already blocked and report blocked state *)
TMP_BLKSTATE := EPCM(DS:RCX).BLOCKED;
```

```
(* at this point, the page must be valid and PT_TCS or PT_REG or PT_TRIM*)
IF (TMP_BLKSTATE = 1)
    THEN
        RFLAGS.CF := 1;
        RAX := SGX_BLKSTATE;
    ELSE
        EPCM(DS:RCX).BLOCKED := 1
FI;
DONE:
```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Sets CF if page is BLOCKED or not blockable, otherwise cleared. Clears PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

ECREATE—Create an SECS page in the Enclave Page Cache

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLS[ECREATE]	IR	V/V	SGX1	This leaf function begins an enclave build by creating an SECS page in EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	ECREATE (In)	Address of a PAGEINFO (In)	Address of the destination SECS page (In)

Description

ENCLS[ECREATE] is the first instruction executed in the enclave build process. ECREATE copies an SECS structure outside the EPC into an SECS page inside the EPC. The internal structure of SECS is not accessible to software.

ECREATE will set up fields in the protected SECS and mark the page as valid inside the EPC. ECREATE initializes or checks unused fields.

Software sets the following fields in the source structure: SECS:BASEADDR, SECS:SIZE in bytes, ATTRIBUTES, CONFIGID, and CONFIGSVN. SECS:BASEADDR must be naturally aligned on an SECS.SIZE boundary. SECS.SIZE must be at least 2 pages (8192).

The source operand RBX contains an effective address of a PAGEINFO structure. PAGEINFO contains an effective address of a source SECS and an effective address of an SECINFO. The SECS field in PAGEINFO is not used.

The RCX register is the effective address of the destination SECS. It is an address of an empty slot in the EPC. The SECS structure must be page aligned. SECINFO flags must specify the page as an SECS page.

ECREATE Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

ECREATE will fault if the SECS target page is in use; already valid; outside the EPC. It will also fault if addresses are not aligned; unused PAGEINFO fields are not zero.

If the amount of space needed to store the SSA frame is greater than the amount specified in SECS.SSAFRAME-SIZE, a #GP(0) results. The amount of space needed for an SSA frame is computed based on DS:TMP_-SECS.ATTRIBUTES.XFRM size. Details of computing the size can be found Section 40.7.

Concurrency Restrictions

Table 39-15. Base Concurrency Restrictions of ECREATE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ECREATE	SECS [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 39-16. Additional Concurrency Restrictions of ECREATE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ECREATE	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ECREATE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the SECS source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the SECS page to be added.
TMP_XSIZE	SSA Size	64	The size calculation of SSA frame.
TMP_MISC_SIZE	MISC Field Size	64	Size of the selected MISC field components.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECINFO := DS:RBX.SECINFO;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECINFO is not 64Byte aligned)
    THEN #GP(0); FI;

IF (DS:RBX.LINADDR ! = 0 or DS:RBX.SECS ≠ 0)
    THEN #GP(0); FI;

(* Check for misconfigured SECINFO flags*)
IF (DS:TMP_SECINFO reserved fields are not zero or DS:TMP_SECINFO.FLAGS.PT ≠ PT_SECS)
    THEN #GP(0); FI;

TMP_SECS := RCX;

IF (EPC entry in use)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address :=
                    << translation of DS:TMP_SECS produced by paging >>;
                VMCS.Guest-linear_address := DS:TMP_SECS;
            Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

IF (EPC entry in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPGE[32767:0];

(* Check lower 2 bits of XFRM are set *)
IF ( ( DS:TMP_SECS.ATTRIBUTES.XFRM BitwiseAND 03H) ≠ 03H)
    THEN #GP(0); FI;

IF (XFRM is illegal)
    THEN #GP(0); FI;

(* Check legality of CET_ATTRIBUTES *)
IF ((DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_ATTRIBUTES ≠ 0) ||
    (DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[5:2] ≠ 0) ||
    (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[1:0] ≠ 0) ||
    (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1 and
     (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) not canonical) ||
    (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0 and
     (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) & 0xFFFFFFFF00000000) ||
    (DS:TMP_SECS.CET_ATTRIBUTES.reserved fields not 0) or
     (DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) is not page aligned))
    THEN
        #GP(0);
FI;

(* Make sure that the SECS does not have any unsupported MISCSELECT options*)
IF ( !(CPUID.(EAX=12H, ECX=0):EBX[31:0] & DS:TMP_SECS.MISCSELECT[31:0]) )
    THEN
        EPCM(DS:TMP_SECS).EntryLock.Release();
        #GP(0);
FI;

( * Compute size of MISC area *)
TMP_MISC_SIZE := compute_misc_region_size();

(* Compute the size required to save state of the enclave on async exit, see Section 40.7.2.2*)
```

```
TMP_XSIZE := compute_xsave_size(DS:TMP_SECS.ATTRIBUTES.XFRM) + GPR_SIZE + TMP_MISC_SIZE;

(* Ensure that the declared area is large enough to hold XSAVE and GPR stat *)
IF ( DS:TMP_SECS.SSAFRAMESIZE*4096 < TMP_XSIZE)
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.BASEADDR is not canonical) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.BASEADDR and 0FFFFFFFF00000000H) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[7:0]) ) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[15:8]) ) )
    THEN #GP(0); FI;

(* Enclave size must be at least 8192 bytes and must be power of 2 in bytes*)
IF (DS:TMP_SECS.SIZE < 8192 or popcnt(DS:TMP_SECS.SIZE) > 1)
    THEN #GP(0); FI;

(* Ensure base address of an enclave is aligned on size*)
IF ( ( DS:TMP_SECS.BASEADDR and (DS:TMP_SECS.SIZE-1) ) )
    THEN #GP(0); FI;

(* Ensure the SECS does not have any unsupported attributes*)
IF ( DS:TMP_SECS.ATTRIBUTES and (~CR_SGX_ATTRIBUTES_MASK) )
    THEN #GP(0); FI;

IF ( DS:TMP_SECS reserved fields are not zero)
    THEN #GP(0); FI;

(* Verify that CONFIGID/CONFIGSVN are not set with attribute *)
IF ( ((DS:TMP_SECS.CONFIGID ≠ 0) or (DS:TMP_SECS.CONFIGSVN ≠0)) AND (DS:TMP_SECS.ATTRIBUTES.KSS == 0 ))
    THEN #GP(0); FI;

Clear DS:TMP_SECS to Uninitialized;
DS:TMP_SECS.MRENCLAVE := SHA256INITIALIZE(DS:TMP_SECS.MRENCLAVE);
DS:TMP_SECS.ISVSVN := 0;
DS:TMP_SECS.ISVPRODID := 0;

(* Initialize hash updates etc*)
Initialize enclave's MRENCLAVE update counter;

(* Add "ECREATE" string and SECS fields to MRENCLAVE *)
TMPUPDATEFIELD[63:0] := 0045544145524345H; // "ECREATE"
TMPUPDATEFIELD[95:64] := DS:TMP_SECS.SSAFRAMESIZE;
TMPUPDATEFIELD[159:96] := DS:TMP_SECS.SIZE;
IF (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 1)
    THEN
        TMPUPDATEFIELD[223:160] := DS:TMP_SECS.CET_LEG_BITMAP_OFFSET;
    ELSE
        TMPUPDATEFIELD[223:160] := 0;
```

```
FI;
TMPUPDATEFIELD[511:160] := 0;
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(* Set EID *)
DS:TMP_SECS.EID := LockedXAdd(CR_NEXT_EID, 1);

(* Initialize the virtual child count to zero *)
DS:TMP_SECS.VIRTCHILDCNT := 0;

(* Load ENCLAVECONTEXT with Address out of paging of SECS *)
<< store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>

(* Set the EPCM entry, first create SECS identifier and store the identifier in EPCM *)
EPCM(DS:TMP_SECS).PT := PT_SECS;
EPCM(DS:TMP_SECS).ENCLAVEADDRESS := 0;
EPCM(DS:TMP_SECS).R := 0;
EPCM(DS:TMP_SECS).W := 0;
EPCM(DS:TMP_SECS).X := 0;

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).VALID := 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the reserved fields are not zero.
	If PAGEINFO.SECS is not zero.
	If PAGEINFO.LINADDR is not zero.
	If the SECS destination is locked.
	If SECS.SSAFRAMESIZE is insufficient.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the SECS destination is outside the EPC.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical form.
	If a memory operand is not properly aligned.
	If the reserved fields are not zero.
	If PAGEINFO.SECS is not zero.
	If PAGEINFO.LINADDR is not zero.
	If the SECS destination is locked.
	If SECS.SSAFRAMESIZE is insufficient.

#PF(error code)	If a page fault occurs in accessing memory operands.
	If the SECS destination is outside the EPC.

EDBGRD—Read From a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLS[EDBGRD]	IR	V/V	SGX1	This leaf function reads a dword/quadword from a debug enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDBGRD (In)	Return error code (Out)	Data read from a debug enclave (Out)	Address of source memory in the EPC (In)

Description

This leaf function copies a quadword/doubleword from an EPC page belonging to a debug enclave into the RBX register. Eight bytes are read in 64-bit mode, four bytes are read in non-64-bit modes. The size of data read cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

EDBGRD Memory Parameter Semantics

EPCQW
Read access permitted by Enclave

The error codes are:

Table 39-17. EDBGRD Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EDBGRD successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

The instruction faults if any of the following:

EDBGRD Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is beyond the architectural size of the TCS (SGX_TCS_LIMIT).
An operand causing any segment violation.	May page fault.
CPL > 0.	

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

Table 39-18. Base Concurrency Restrictions of EDBGRD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGRD	Target [DS:RCX]	Shared	#GP	

Table 39-19. Additional Concurrency Restrictions of EDBGRD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGRD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDBGRD Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1))
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing the same EPCM entry *)
IF (Another instruction modifying the same EPCM entry is executing)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (SOURCE) is pointing to a PT_REG or PT_TCS or PT_VA or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_VA)
and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)
IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_DEBUGGABLE;
        GOTO DONE;
FI;

(* If source is a TCS, then make sure that the offset into the page is not beyond the TCS size*)
IF ( ( EPCM(DS:RCX). PT = PT_TCS) and ((DS:RCX) & FFFH ≥ SGX_TCS_LIMIT) )
    THEN #GP(0); FI;

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF ( (EPCM(DS:RCX).PT = PT_REG) or (EPCM(DS:RCX).PT = PT_TCS) )
    THEN
        TMP_SECS := GET_SECS_ADDRESS;
        IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
            THEN #GP(0); FI;
        IF ( (TMP_MODE64 = 1) )
            THEN RBX[63:0] := (DS:RCX)[63:0];
            ELSE EBX[31:0] := (DS:RCX)[31:0];
        FI;
    ELSE
        TMP_64BIT_VAL[63:0] := (DS:RCX)[63:0] & (~07H); // Read contents from VA slot
        IF (TMP_MODE64 = 1)
            THEN
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN RBX[63:0] := 0FFFFFFFFFFFFFFFH;
                    ELSE RBX[63:0] := 0H;
                FI;
            ELSE
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN EBX[31:0] := 0FFFFFFFFH;
                    ELSE EBX[31:0] := 0H;
                FI;
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set to 0. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights.
	If DS segment is unusable.
	If RCX points to a memory location not 4Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
	If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.

#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RCX is non-canonical form.
	If RCX points to a memory location not 8Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
	If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EDBGWR—Write to a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLS[EDBGWR]	IR	V/V	SGX1	This leaf function writes a dword/quadword to a debug enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDBGWR (In)	Return error code (Out)	Data to be written to a debug enclave (In)	Address of Target memory in the EPC (In)

Description

This leaf function copies the content in EBX/RBX to an EPC page belonging to a debug enclave. Eight bytes are written in 64-bit mode, four bytes are written in non-64-bit modes. The size of data cannot be overridden.

The effective address of the target location inside the EPC is provided in the register RCX.

EDBGWR Memory Parameter Semantics

EPCQW
Write access permitted by Enclave

The instruction faults if any of the following:

EDBGWR Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is not the FLAGS word.
An operand causing any segment violation.	May page fault.
CPL > 0.	

The error codes are:

Table 39-20. EDBGWR Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EDBGWR successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

Table 39-21. Base Concurrency Restrictions of EDBGWR

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGWR	Target [DS:RCX]	Shared	#GP	

Table 39-22. Additional Concurrency Restrictions of EDBGWR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGWR	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDBGWR Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing the same EPCM entry *)
IF (Another instruction modifying the same EPCM entry is executing)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS)
  and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)
IF ( (EPCM(DS:RCX).PENDING is not 0) or (EPCM(DS:RCS).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_DEBUGGABLE;
        GOTO DONE;
FI;

(* If destination is a TCS, then make sure that the offset into the page can only point to the FLAGS field*)
IF ( ( EPCM(DS:RCX). PT = PT_TCS) and ((DS:RCX) & FF8H ≠ offset_of_FLAGS & 0FF8H) )
    THEN #GP(0); FI;

(* Locate the SECS for the enclave to which the DS:RCX page belongs *)
TMP_SECS := GET_SECS_PHYS_ADDRESS(EPCM(DS:RCX).ENCLAVESECS);

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 1) )
    THEN (DS:RCX)[63:0] := RBX[63:0];
    ELSE (DS:RCX)[31:0] := EBX[31:0];
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0
```

Flags Affected

ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set to 0. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights.
	If DS segment is unusable.
	If RCX points to a memory location not 4Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RCX is non-canonical form.
	If RCX points to a memory location not 8Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.

#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLS[EEXTEND]	IR	V/V	SGX1	This leaf function measures 256 bytes of an uninitialized enclave page.

Instruction Operand Encoding

Op/En	EAX	EBX	RCX
IR	EEXTEND (In)	Effective address of the SECS of the data chunk (In)	Effective address of a 256-byte chunk in the EPC (In)

Description

This leaf function updates the MRENCLAVE measurement register of an SECS with the measurement of an EXTEND string compromising of "EEXTEND" || ENCLAVEOFFSET || PADDING || 256 bytes of the enclave page. This instruction can only be executed when current privilege level is 0 and the enclave is uninitialized.

RBX contains the effective address of the SECS of the region to be measured. The address must be the same as the one used to add the page into the enclave.

RCX contains the effective address of the 256 byte region of an EPC page to be measured. The DS segment is used to create linear addresses. Segment override is not supported.

EEXTEND Memory Parameter Semantics

EPC[RCX]
Read access by Enclave

The instruction faults if any of the following:

EEXTEND Faulting Conditions

RBX points to an address not 4KBytes aligned.	RBX does not resolve to an SECS.
RBX does not point to an SECS page.	RBX does not point to the SECS page of the data chunk.
RCX points to an address not 256B aligned.	RCX points to an unused page or a SECS.
RCX does not resolve in an EPC page.	If SECS is locked.
If the SECS is already initialized.	May page fault.
CPL > 0.	

Concurrency Restrictions

Table 39-23. Base Concurrency Restrictions of EEXTEND

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXTEND	Target [DS:RCX]	Shared	#GP	
	SECS [DS:RBX]	Concurrent		

Table 39-24. Additional Concurrency Restrictions of EEXTEND

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXTEND	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EEXTEND Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (DS:RBX is not 4096 Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve to an EPC page)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX is not 256Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other instructions accessing EPCM)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS)
 and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
    THEN #PF(DS:RCX); FI;

TMP_SECS := Get_SECS_ADDRESS();

IF (DS:RBX does not resolve to TMP_SECS)
    THEN #GP(0); FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction accessing MRENCLAVE) or (Other instructions checking or updating the initialized state of the SECS))
```

```
    THEN #GP(0); FI;

(* Calculate enclave offset *)
TMP_ENCLAVEOFFSET := EPCM(DS:RCX).ENCLAVEADDRESS - TMP_SECS.BASEADDR;
TMP_ENCLAVEOFFSET := TMP_ENCLAVEOFFSET + (DS:RCX & 0FFFH)

(* Add EEXTEND message and offset to MRENCLAVE *)
TMPUPDATEFIELD[63:0] := 00444E4554584545H; // "EEXTEND"
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] := 0; // 48 bytes
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(*Add 256 bytes to MRENCLAVE, 64 byte at a time *)
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[511:0] );
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1023: 512] );
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1535: 1024] );
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[2047: 1536] );
INC enclave's MRENCLAVE update counter by 4;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RBX is outside the DS segment limit.
	If RBX points to an SECS page which is not the SECS of the data chunk.
	If the address in RCX is outside the DS segment limit.
	If RCX points to a memory location not 256Byte-aligned.
	If another instruction is accessing MRENCLAVE.
	If another instruction is checking or updating the SECS.
	If the enclave is already initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RBX points to a non-EPC page.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RBX is non-canonical form.
	If RBX points to an SECS page which is not the SECS of the data chunk.
	If RCX is non-canonical form.
	If RCX points to a memory location not 256 Byte-aligned.
	If another instruction is accessing MRENCLAVE.
	If another instruction is checking or updating the SECS.
	If the enclave is already initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RBX points to a non-EPC page.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EINIT—Initialize an Enclave for Execution

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLS[EINIT]	IR	V/V	SGX1	This leaf function initializes the enclave and makes it ready to execute enclave code.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EINIT (In)	Error code (Out)	Address of SIGSTRUCT (In)	Address of SECS (In)	Address of EINITTOKEN (In)

Description

This leaf function is the final instruction executed in the enclave build process. After EINIT, the MRENCLAVE measurement is complete, and the enclave is ready to start user code execution using the EENTER instruction.

EINIT takes the effective address of a SIGSTRUCT and EINITTOKEN. The SIGSTRUCT describes the enclave including MRENCLAVE, ATTRIBUTES, ISVSVN, a 3072 bit RSA key, and a signature using the included key. SIGSTRUCT must be populated with two values, q1 and q2. These are calculated using the formulas shown below:

q1 = floor(Signature^2 / Modulus);

q2 = floor((Signature^3 - q1 * Signature * Modulus) / Modulus);

The EINITTOKEN contains the MRENCLAVE, MRSIGNER, and ATTRIBUTES. These values must match the corresponding values in the SECS. If the EINITTOKEN was created with a debug launch key, the enclave must be in debug mode as well.

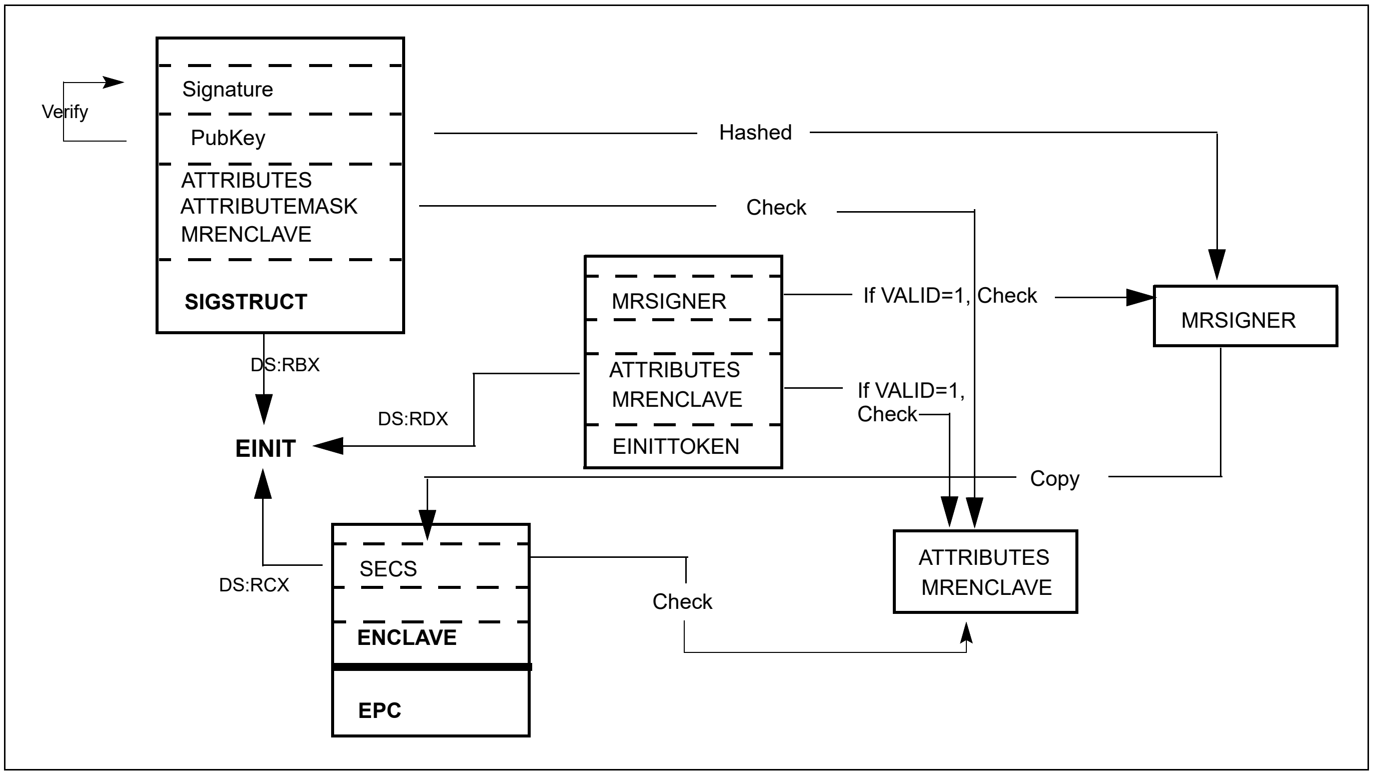


Figure 39-1. Relationships Between SECS, SIGSTRUCT, and EINITTOKEN

EINIT Memory Parameter Semantics

SIGSTRUCT	SECS	EINITTOKEN
Access by non-Enclave	Read/Write access by Enclave	Access by non-Enclave

EINIT performs the following steps, which can be seen in Figure 39-1:

1. Validates that SIGSTRUCT is signed using the enclosed public key.
2. Checks that the completed computation of SECS.MRENCLAVE equals SIGSTRUCT.HASHENCLAVE.
3. Checks that no controlled ATTRIBUTES bits are set in SIGSTRUCT.ATTRIBUTES unless the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.
4. Checks that the result of bitwise and-ing SIGSTRUCT.ATTRIBUTEMASK with SIGSTRUCT.ATTRIBUTES equals the result of bitwise and-ing SIGSTRUCT.ATTRIBUTEMASK with SECS.ATTRIBUTES.
5. If EINITTOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUB-KEYHASH.
6. If EINITTOKEN.VALID is 1, checks the validity of EINITTOKEN.
7. If EINITTOKEN.VALID is 1, checks that EINITTOKEN.MRENCLAVE equals SECS.MRENCLAVE.
8. If EINITTOKEN.VALID is 1 and EINITTOKEN.ATTRIBUTES.DEBUG is 1, SECS.ATTRIBUTES.DEBUG must be 1.
9. Commits SECS.MRENCLAVE, and sets SECS.MRSIGNER, SECS.ISVSVN, and SECS.ISVPRODID based on SIGSTRUCT.
10. Update the SECS as Initialized.

Periodically, EINIT polls for certain asynchronous events. If such an event is detected, it completes with failure code (ZF=1 and RAX = SGX_UNMASKED_EVENT), and RIP is incremented to point to the next instruction. These events includes external interrupts, non-maskable interrupts, system-management interrupts, machine checks, INIT signals, and the VMX-preemption timer. EINIT does not fail if the pending event is inhibited (e.g., external interrupts could be inhibited due to blocking by MOV SS blocking or by STI).

The following bits in RFLAGS are cleared: CF, PF, AF, OF, and SF. When the instruction completes with an error, RFLAGS.ZF is set to 1, and the corresponding error bit is set in RAX. If no error occurs, RFLAGS.ZF is cleared and RAX is set to 0.

The error codes are:

Table 39-25. EINIT Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EINIT successful.
SGX_INVALID_SIG_STRUCT	If SIGSTRUCT contained an invalid value.
SGX_INVALID_ATTRIBUTE	If SIGSTRUCT contains an unauthorized attributes mask.
SGX_INVALID_MEASUREMENT	If SIGSTRUCT contains an incorrect measurement. If EINITTOKEN contains an incorrect measurement.
SGX_INVALID_SIGNATURE	If signature does not validate with enclosed public key.
SGX_INVALID_LICENSE	If license is invalid.
SGX_INVALID_CPUSVN	If license SVN is unsupported.
SGX_UNMASKED_EVENT	If an unmasked event is received before the instruction completes its operation.

Concurrency Restrictions

Table 39-26. Base Concurrency Restrictions of EINIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINIT	SECS [DS:RCX]	Shared	#GP	

Table 39-27. Additional Concurrency Restrictions of ENIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINIT	SECS [DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EINIT Operational Flow

Name	Type	Size	Description
TMP_SIG	SIGSTRUCT	1808Bytes	Temp space for SIGSTRUCT.
TMP_TOKEN	EINITTOKEN	304Bytes	Temp space for EINITTOKEN.
TMP_MRENCLAVE		32Bytes	Temp space for calculating MRENCLAVE.
TMP_MRSIGNER		32Bytes	Temp space for calculating MRSIGNER.
CONTROLLED_ATTRIBUTES	ATTRIBUTES	16Bytes	Constant mask of all ATTRIBUTE bits that can only be set for authorized enclaves.
TMP_KEYDEPENDENCIES	Buffer	224Bytes	Temp space for key derivation.
TMP_EINITTOKENKEY		16Bytes	Temp space for the derived EINITTOKEN Key.
TMP_SIG_PADDING	PKCS Padding Buffer	352Bytes	The value of the top 352 bytes from the computation of Signature[3] modulo MRSIGNER.

```
(* make sure SIGSTRUCT and SECS are aligned *)
IF ( (DS:RBX is not 4KByte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* make sure the EINITTOKEN is aligned *)
IF (DS:RDX is not 512Byte Aligned)
    THEN #GP(0); FI;

(* make sure the SECS is inside the EPC *)
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SIG[14463:0] := DS:RBX[14463:0]; // 1808 bytes
TMP_TOKEN[2423:0] := DS:RDX[2423:0]; // 304 bytes
```

```
(* Verify SIGSTRUCT Header. *)
IF ( (TMP_SIG.HEADER ≠ 06000000E10000000000010000000000h) or
    ((TMP_SIG.VENDOR ≠ 0) and (TMP_SIG.VENDOR ≠ 00008086h) ) or
    (TMP_SIG HEADER2 ≠ 01010000600000006000000001000000h) or
    (TMP_SIG.EXPONENT   ≠ 00000003h) or (Reserved space is not 0's) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_SIG_STRUCT;
        GOTO EXIT;
FI;

(* Open "Event Window" Check for Interrupts. Verify signature using embedded public key, q1, and q2. Save upper 352 bytes of the
PKCS1.5 encoded message into the TMP_SIG_PADDING*)
IF (interrupt was pending) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_UNMASKED_EVENT;
    GOTO EXIT;
FI
IF (signature failed to verify) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_SIGNATURE;
    GOTO EXIT;
FI;
(*Close "Event Window" *)

(* make sure no other Intel SGX instruction is modifying SECS*)
IF (Other instructions modifying SECS)
    THEN #GP(0); FI;

IF ( (EPCM(DS:RCX). VALID = 0) or (EPCM(DS:RCX).PT ≠ PT_SECS) )
    THEN #PF(DS:RCX); FI;

(* Verify ISVFAMILYID is not used on an enclave with KSS disabled *)
IF ((TMP_SIG.ISVFAMILYID != 0) AND (DS:RCX.ATTRIBUTES.KSS == 0))
    THEN
     RFLAGS.ZF := 1;
     RAX := SGX_INVALID_SIG_STRUCT;
     GOTO EXIT;
FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction modifying MRENCLAVE) or (Other instructions modifying the SECS's Initialized state))
    THEN #GP(0); FI;

(* Calculate finalized version of MRENCLAVE *)
(* SHA256 algorithm requires one last update that compresses the length of the hashed message into the output SHA256 digest *)
TMP_ENCLAVE := SHA256FINAL( (DS:RCX).MRENCLAVE, enclave's MRENCLAVE update count *512);

(* Verify MRENCLAVE from SIGSTRUCT *)
IF (TMP_SIG.ENCLAVEHASH ≠ TMP_MRENCLAVE)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;
```

```
TMP_MRSIGNER := SHA256(TMP_SIG.MODULUS)

(* if controlled ATTRIBUTES are set, SIGSTRUCT must be signed using an authorized key *)
CONTROLLED_ATTRIBUTES := 0000000000000020H;
IF ( ( (DS:RCX.ATTRIBUTES & CONTROLLED_ATTRIBUTES) ≠ 0) and (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH) )
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_ATTRIBUTE;
    GOTO EXIT;
FI;

(* Verify SIGSTRUCT.ATTRIBUTE requirements are met *)
IF ( (DS:RCX.ATTRIBUTES & TMP_SIG.ATTRIBUTEMASK) ≠ (TMP_SIG.ATTRIBUTE & TMP_SIG.ATTRIBUTEMASK) )
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_ATTRIBUTE;
    GOTO EXIT;
FI;

( *Verify SIGSTRUCT.MISCSELECT requirements are met *)
IF ( (DS:RCX.MISCSELECT & TMP_SIG.MISCMASK) ≠ (TMP_SIG.MISCSELECT & TMP_SIG.MISCMASK) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
    GOTO EXIT
FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    IF ( DS:RCX.CET_ATTRIBUTES & TMP_SIG.CET_ATTRIBUTES_MASK ≠ TMP_SIG.CET_ATTRIBUTES &
     TMP_SIG.CET_ATTRIB-UTES_MASK )
        THEN
            RFLAGS.ZF := 1;
            RAX := SGX_INVALID_ATTRIBUTE;
            GOTO EXIT
    FI;
FI;

(* If EINITTOKEN.VALID[0] is 0, verify the enclave is signed by an authorized key *)
IF (TMP_TOKEN.VALID[0] = 0)
    IF (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH)
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_EINITTOKEN;
        GOTO EXIT;
    FI;
    GOTO COMMIT;
FI;

(* Debug Launch Enclave cannot launch Production Enclaves *)
IF ( (DS:RDX.MASKEDATTRIBUTESLE.DEBUG = 1) and (DS:RCX.ATTRIBUTES.DEBUG = 0) )
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;
```

```
(* Check reserve space in EINIT token includes reserved regions and upper bits in valid field *)
IF (TMP_TOKEN reserved space is not clear)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* EINIT token must not have been created by a configuration beyond the current CPU configuration *)
IF (TMP_TOKEN.CPUSVN must not be a configuration beyond CR_CPUSVN)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_CPUSVN;
    GOTO EXIT;
FI;

(* Derive Launch key used to calculate EINITTOKEN.MAC *)
HARDCODED_PKCS1_5_PADDING[15:0] := 0100H;
HARDCODED_PKCS1_5_PADDING[2655:16] := SignExtend330Byte(-1); // 330 bytes of 0FFH
HARDCODED_PKCS1_5_PADDING[2815:2656] := 2004000501020403650148866009060D30313000H;

TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_TOKEN.ISVPRODIDLE;
TMP_KEYDEPENDENCIES.ISVSVN := TMP_TOKEN.ISVSVNLE;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_TOKEN.MASKEDATTRIBUTESLE;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := IA32_SGXLEPUBKEYHASH;
TMP_KEYDEPENDENCIES.KEYID := TMP_TOKEN.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := TMP_TOKEN.CPUSVNLE;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_TOKEN.MASKEDMISCSELECTLE;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_TOKEN.CET_MASKED_ATTRIBUTES_ LE;
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;

(* Calculate the derived key*)
TMP_EINITTOKENKEY := derivekey(TMP_KEYDEPENDENCIES);

(* Verify EINITTOKEN was generated using this CPU's Launch key and that it has not been modified since issuing by the Launch
Enclave. Only 192 bytes of EINITTOKEN are CMACed *)
IF (TMP_TOKEN.MAC ≠ CMAC(TMP_EINITTOKENKEY, TMP_TOKEN[1535:0] ) )
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;
```

```
(* Verify EINITTOKEN (RDX) is for this enclave *)
IF ( (TMP_TOKEN.MRENCLAVE ≠ TMP_MRENCLAVE) or (TMP_TOKEN.MRSIGNER ≠ TMP_MRSIGNER) )
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

(* Verify ATTRIBUTES in EINITTOKEN are the same as the enclave's *)
IF (TMP_TOKEN.ATTRIBUTES ≠ DS:RCX.ATTRIBUTES)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINIT_ATTRIBUTE;
    GOTO EXIT;
FI;

COMMIT:
(* Commit changes to the SECS; Set ISVPRODID, ISVSVN, MRSIGNER, INIT ATTRIBUTE fields in SECS (RCX) *)
DS:RCX.MRENCLAVE := TMP_MRENCLAVE;
(* MRSIGNER stores a SHA256 in little endian implemented natively on x86 *)
DS:RCX.MRSIGNER := TMP_MRSIGNER;
DS:RCX.ISVEXTPRODID := TMP_SIG.ISVEXTPRODID;
DS:RCX.ISVPRODID := TMP_SIG.ISVPRODID;
DS:RCX.ISVSVN := TMP_SIG.ISVSVN;
DS:RCX.ISVFAMILYID := TMP_SIG.ISVFAMILYID;
DS:RCX.PADDING := TMP_SIG_PADDING;

(* Mark the SECS as initialized *)
Update DS:RCX to initialized;

(* Set RAX and ZF for success*)
    RFLAGS.ZF := 0;
    RAX := 0;
EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set and RAX contains the error code. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand is not properly aligned.
	If another instruction is modifying the SECS.
	If the enclave is already initialized.
	If the SECS.MRENCLAVE is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RCX does not resolve in an EPC page.
	If the memory address is not a valid, uninitialized SECS.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is not properly aligned.
	If another instruction is modifying the SECS.
	If the enclave is already initialized.
	If the SECS.MRENCLAVE is in use.

#PF(error code)	If a page fault occurs in accessing memory operands.
	If RCX does not resolve in an EPC page.
	If the memory address is not a valid, uninitialized SECS.

ELDB/ELDU/ELDBC/ELDUC—Load an EPC Page and Mark its State

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLS[ELDB]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as blocked.
EAX = 08H ENCLS[ELDU]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as unblocked.
EAX = 12H ENCLS[ELDBC]	IR	V/V	EAX[6]	This leaf function behaves lie ELDB but with improved conflict handling for oversubscription.
EAX = 13H ENCLS[ELDUC]	IR	V/V	EAX[6]	This leaf function behaves like ELDU but with improved conflict handling for oversubscription.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	ELDB/ELDU (In)	Return error code (Out)	Address of the PAGEINFO (In)	Address of the EPC page (In)	Address of the version-array slot (In)

Description

This leaf function copies a page from regular main memory to the EPC. As part of the copying process, the page is cryptographically authenticated and decrypted. This instruction can only be executed when current privilege level is 0.

The ELDB leaf function sets the BLOCK bit in the EPCM entry for the destination page in the EPC after copying. The ELDU leaf function clears the BLOCK bit in the EPCM entry for the destination page in the EPC after copying.

RBX contains the effective address of a PAGEINFO structure; RCX contains the effective address of the destination EPC page; RDX holds the effective address of the version array slot that holds the version of the page.

The ELDBC/ELDUC leafs are very similar to ELDB and ELDU. They provide an error code on the concurrency conflict for any of the pages which need to acquire a lock. These include the destination, SECS, and VA slot.

The table below provides additional information on the memory parameter of ELDB/ELDU leaf functions.

ELDB/ELDU/ELDBC/ELBUC Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	PAGEINFO.SECS	EPCPAGE	Version-Array Slot
Non-enclave read access	Non-enclave read access	Non-enclave read access	Enclave read/write access	Read/Write access permitted by Enclave	Read/Write access permitted by Enclave

The error codes are:

Table 39-28. ELDB/ELDU/ELDBC/ELBUC Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	ELDB/ELDU successful.
SGX_MAC_COMPARE_FAIL	If the MAC check fails.

Concurrency Restrictions

Table 39-29. Base Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ELDB/ELDU	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	
ELDBC/ELBUC	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_ CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA [DS:RDX]	Shared	SGX_EPC_PAGE_ CONFLICT	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	SGX_EPC_PAGE_ CONFLICT	

Table 39-30. Additional Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ELDB/ELDU	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	
ELDBC/ELBUC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ELDB/ELDU/ELDBC/ELBUC Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Memory page	4KBytes	
TMP_SECS	Memory page	4KBytes	
TMP_PCMD	PCMD	128 Bytes	
TMP_HEADER	MACHEADER	128 Bytes	
TMP_VER	UINT64	64	
TMP_MAC	UINT128	128	
TMP_PK	UINT128	128	Page encryption/MAC key.
SCRATCH_PCMD	PCMD	128 Bytes	

```
(* Check PAGEINFO and EPCPAGE alignment *)
IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check VASLOT alignment *)
IF (DS:RDX is not 8Byte aligned)
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECS := DS:RBX.SECS;
TMP_PCMD := DS:RBX.PCMD;

(* Check alignment of PAGEINFO (RBX) linked parameters. Note: PCMD pointer is overlaid on top of PAGEINFO.SECINFO field *)
IF ( (DS:TMP_PCMD is not 128Byte aligned) or (DS:TMP_SRCPGE is not 4KByte aligned) )
    THEN #GP(0); FI;

(* Check concurrency of EPC by other Intel SGX instructions *)
IF (other instructions accessing EPC)
    THEN
    IF ((EAX==07h) OR (EAX==08h))   (* ELDB/ELDU *)
        THEN
            IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                THEN
                    VMCS.Exit_reason := SGX_CONFLICT;
                    VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                    VMCS.Exit_qualification.error := 0;
                    VMCS.Guest-physical_address :=
                << translation of DS:RCX produced by paging >>;
                    VMCS.Guest-linear_address := DS:RCX;
                    Deliver VMEXIT;
                ELSE
                    #GP(0);
                FI;
            ELSE (* ELDBC/ELDUC *)
            IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                THEN
                    VMCS.Exit_reason := SGX_CONFLICT;
                    VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_ERROR;
                    VMCS.Exit_qualification.error := SGX_EPC_PAGE_CONFLICT;
                    VMCS.Guest-physical_address :=
                << translation of DS:RCX produced by paging >>;
                    VMCS.Guest-linear_address := DS:RCX;
                    Deliver VMEXIT;
                ELSE
                RFLAGS.ZF := 1;
                  RFLAGS.CF := 0;
                RAX := SGX_EPC_PAGE_CONFLICT;
                GOTO ERROR_EXIT;
                FI;
        FI;
FI;

(* Check concurrency of EPC and VASLOT by other Intel SGX instructions *)
IF (Other instructions modifying VA slot) THEN
    IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
        THEN #GP(0);
    ELSE (* ELDBC/ELDUC *)
        RFLAGS.ZF := 1;
        RFLAGS.CF := 0;
        RAX := SGX_EPC_PAGE_CONFLICT;
        GOTO ERROR_EXIT;
    FI;
FI;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~0FFFH).PT ≠ PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Copy PCMD into scratch buffer *)
SCRATCH_PCMD[1023: 0] := DS:TMP_PCMD[1023:0];

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER)-1: 0] := 0;

TMP_HEADER.SECINFO := SCRATCH_PCMD.SECINFO;
TMP_HEADER.RSVD := SCRATCH_PCMD.RSVD;
TMP_HEADER.LINADDR := DS:RBX.LINADDR;

(* Verify various attributes of SECS parameter *)
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
    THEN
        IF ( DS:TMP_SECS is not 4KByte aligned)
            THEN #GP(0) FI;
        IF (DS:TMP_SECS does not resolve within an EPC)
            THEN #PF(DS:TMP_SECS) FI;
        IF ( Another instruction is currently modifying the SECS) THEN
            IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
                THEN #GP(0);
            ELSE (* ELDBC/ELDUC *)
                RFLAGS.ZF := 1;
                RFLAGS.CF := 0;
                RAX := SGX_EPC_PAGE_CONFLICT;
                GOTO ERROR_EXIT;
            FI;
        FI;
        TMP_HEADER.EID := DS:TMP_SECS.EID;
    ELSE
```

```
        (* TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS or PT_VA which do not have a parent SECS, and hence no EID binding *)
        TMP_HEADER.EID := 0;
        IF (DS:TMP_SECS ≠ 0)
            THEN #GP(0) FI;
FI;

(* Copy 4KBytes SRCPGE to secure location *)
DS:RCX[32767: 0] := DS:TMP_SRCPGE[32767: 0];
TMP_VER := DS:RDX[63:0];

(* Decrypt and MAC page. AES_GCM_DEC has 2 outputs, {plain text, MAC} *)
(* Parameters for AES_GCM_DEC {Key, Counter, ..} *)
{DS:RCX, TMP_MAC} := AES_GCM_DEC(CR_BASE_PK, TMP_VER << 32, TMP_HEADER, 128, DS:RCX, 4096);

IF ( (TMP_MAC ≠ DS:TMP_PCMD.MAC) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_MAC_COMPARE_FAIL;
        GOTO ERROR_EXIT;
FI;

(* Clear VA Slot *)
DS:RDX := 0

(* Commit EPCM changes *)
EPCM(DS:RCX).PT := TMP_HEADER.SECINFO.FLAGS.PT;
EPCM(DS:RCX).RWX := TMP_HEADER.SECINFO.FLAGS.RWX;
EPCM(DS:RCX).PENDING := TMP_HEADER.SECINFO.FLAGS.PENDING;
EPCM(DS:RCX).MODIFIED := TMP_HEADER.SECINFO.FLAGS.MODIFIED;
EPCM(DS:RCX).PR := TMP_HEADER.SECINFO.FLAGS.PR;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_HEADER.LINADDR;

IF ( ((EAX = 07H) or (EAX = 12H)) and (TMP_HEADER.SECINFO.FLAGS.PT is NOT PT_SECS or PT_VA))
    THEN
        EPCM(DS:RCX).BLOCKED := 1;
    ELSE
        EPCM(DS:RCX).BLOCKED := 0;
FI;

IF (TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS)
    << store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
FI;

EPCM(DS:RCX). VALID := 1;

RAX := 0;
RFLAGS.ZF := 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the instruction’s EPC resource is in use by others.
	If the instruction fails to verify MAC.
	If the version-array slot is in use.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand expected to be in EPC does not resolve to an EPC page.
	If one of the EPC memory operands has incorrect page type.
	If the destination EPC page is already valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the instruction’s EPC resource is in use by others.
	If the instruction fails to verify MAC.
	If the version-array slot is in use.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand expected to be in EPC does not resolve to an EPC page.
	If one of the EPC memory operands has incorrect page type.
	If the destination EPC page is already valid.

EMODPR—Restrict the Permissions of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0EH ENCLS[EMODPR]	IR	V/V	SGX2	This leaf function restricts the access rights associated with a EPC page in an initialized enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODPR (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function restricts the access rights associated with an EPC page in an initialized enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not restrict the page permissions will have no effect. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPR leaf function.

EMODPR Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

EMODPR Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

Table 39-31. EMODPR Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EMODPR successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

Concurrency Restrictions

Table 39-32. Base Concurrency Restrictions of EMODPR

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPR	Target [DS:RCX]	Shared	#GP	

Table 39-33. Additional Concurrency Restrictions of EMODPR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPR	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

Operation

Temp Variables in EMODPR Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or
    (SCRATCH_SECINFO.FLAGS.R is 0 and SCRATCH_SECINFO.FLAGS.W is not 0) )
    THEN #GP(0); FI;

(* Check concurrency with SGX1 or SGX2 instructions on the EPC page *)
IF (SGX1 or other SGX2 instructions accessing EPC page)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID is 0 )
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT is not PT_REG)
    THEN #PF(DS:RCX); FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
  THEN #GP(0); FI;

(* Set the PR bit to indicate that permission restriction is in progress *)
EPCM(DS:RCX).PR := 1;

(* Update EPCM permissions *)
EPCM(DS:RCX).R := EPCM(DS:RCX).R & SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := EPCM(DS:RCX).W & SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := EPCM(DS:RCX).X & SCRATCH_SECINFO.FLAGS.X;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

EMODT—Change the Type of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0FH ENCLS[EMODT]	IR	V/V	SGX2	This leaf function changes the type of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function modifies the type of an EPC page. The security attributes are configured to prevent access to the EPC page at its new type until a corresponding invocation of the EACCEPT leaf confirms the modification. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODT leaf function.

EMODT Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

EMODT Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

Table 39-34. EMODT Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EMODT successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODPR, or EWB.

Concurrency Restrictions

Table 39-35. Base Concurrency Restrictions of EMODT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_ CONFLICT	EPC_PAGE_CONFLICT_ERROR

Table 39-36. Additional Concurrency Restrictions of EMODT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

Operation

Temp Variables in EMODT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or
    !(SCRATCH_SECINFO.FLAGS.PT is PT_TCS or SCRATCH_SECINFO.FLAGS.PT is PT_TRIM) )
    THEN #GP(0); FI;

(* Check concurrency with SGX1 instructions on the EPC page *)
IF (other SGX1 instructions accessing EPC page)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).VALID is 0)
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
```

```
FI;

IF (!(EPCM(DS:RCX).PT is PT_REG or
    ((EPCM(DS:RCX).PT is PT_TCS or PT_SS_FIRST or PT_SS_REST) and SCRATCH_SECINFO.FLAGS.PT is PT_TRIM)))
        THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Update EPCM fields *)
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).MODIFIED := 1;
EPCM(DS:RCX).R := 0;
EPCM(DS:RCX).W := 0;
EPCM(DS:RCX).X := 0;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

EPA—Add Version Array

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0AH ENCLS[EPA]	IR	V/V	SGX1	This leaf function adds a Version Array to the EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EPA (In)	PT_VA (In, Constant)	Effective address of the EPC page (In)

Description

This leaf function creates an empty version array in the EPC page whose logical address is given by DS:RCX, and sets up EPCM attributes for that page. At the time of execution of this instruction, the register RBX must be set to PT_VA.

The table below provides additional information on the memory parameter of EPA leaf function.

EPA Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

Concurrency Restrictions

Table 39-37. Base Concurrency Restrictions of EPA

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EPA	VA [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 39-38. Additional Concurrency Restrictions of EPA

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EPA	VA [DS:RCX]	Concurrent	L	Concurrent		Concurrent	

Operation

```
IF (RBX ≠ PT_VA or DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions accessing the page)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
```

```
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address := DS:RCX;
            Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

(* Check EPC page must be empty *)
IF (EPCM(DS:RCX). VALID ≠ 0)
   THEN #PF(DS:RCX); FI;

(* Clears EPC page *)
DS:RCX[32767:0] := 0;

EPCM(DS:RCX).PT := PT_VA;
EPCM(DS:RCX).ENCLAVEADDRESS := 0;
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).RWX := 0;
EPCM(DS:RCX).VALID := 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If the EPC page is valid.

ERDINFO—Read Type and Status Information About an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 10H ENCLS[ERDINFO]	IR	V/V	EAX[6]	This leaf function returns type and status information about an EPC page.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	ERDINFO (In)	Return error code (Out)	Address of a RDINFO structure (In)	Address of the destination EPC page (In)

Description

This instruction reads type and status information about an EPC page and returns it in a RDINFO structure. The STATUS field of the structure describes the status of the page and determines the validity of the remaining fields. The FLAGS field returns the EPCM permissions of the page; the page type; and the BLOCKED, PENDING, MODIFIED, and PR status of the page. For enclave pages, the ENCLAVECONTEXT field of the structure returns the value of SECS.ENCLAVECONTEXT. For non-enclave pages (e.g., VA) ENCLAVECONTEXT returns 0.

For invalid or non-EPC pages, the instruction returns an information code indicating the page's status, in addition to populating the STATUS field.

ERDINFO returns an error code if the destination EPC page is being modified by a concurrent SGX instruction.

RBX contains the effective address of a RDINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of ERDINFO leaf function.

ERDINFO Memory Parameter Semantics

RDINFO	EPCPAGE
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

ERDINFO Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

The error codes are:

Table 39-39. ERDINFO Return Value in RAX

Error Code	Value	Description
No Error	0	ERDINFO successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD		Target page is not a valid EPC page.
SGX_PG_NONEPC		Page is not an EPC page.

Concurrency Restrictions

Table 39-40. Base Concurrency Restrictions of ERDINFO

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERDINFO	Target [DS:RCX]	Shared	SGX_EPC_PAGE_ CONFLICT	

Table 39-41. Additional Concurrency Restrictions of ERDINFO

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERDINFO	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ERDINFO Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_RDINFO	Linear Address	64	Address of the RDINFO structure.

```
(* check alignment of RDINFO structure (RBX) *)
IF (DS:RBX is not 32Byte Aligned) THEN
    #GP(0); FI;

(* check alignment of the EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); FI;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within EPC) THEN
    RFLAGS.CF := 1;
    RFLAGS.ZF := 0;
    RAX := SGX_PG_NONEPC;
    goto DONE;
FI;

(* Check the EPC page for concurrency *)
IF (EPC page is being modified) THEN
    RFLAGS.ZF = 1;
    RFLAGS.CF = 0;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check page validity *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    RFLAGS.CF = 1;
    RFLAGS.ZF = 0;
    RAX = SGX_PG_INVLD;
    goto DONE;
FI;

(* clear the fields of the RDINFO structure *)
TMP_RDINFO := DS:RBX;
TMP_RDINFO.STATUS := 0;
TMP_RDINFO.FLAGS := 0;
TMP_RDINFO.ENCLAVECONTEXT := 0;

(* store page info in RDINFO structure *)
TMP_RDINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
TMP_RDINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_RDINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_RDINFO.FLAGS.PR := EPCM(DS:RCX).PR;
TMP_RDINFO.FLAGS.PAGE_TYPE := EPCM(DS:RCX).PAGE_TYPE;
TMP_RDINFO.FLAGS.BLOCKED := EPCM(DS:RCX).BLOCKED;

(* read SECS.ENCLAVECONTEXT for enclave child pages *)
IF ((EPCM(DS:RCX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_REST)
   ) THEN
    TMP_SECS := Address of SECS for (DS:RCX);
    TMP_RDINFO.ENCLAVECONTEXT := SECS(TMP_SECS).ENCLAVECONTEXT;
FI;

(* populate enclave information for SECS pages *)
IF (EPCM(DS:RCX).PAGE_TYPE = PT_SECS) THEN
    IF ((VMX non-root mode) and
        (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)
       ) THEN
        TMP_RDINFO.STATUS.CHILDPRESENT :=
                    ((SECS(DS:RCX).CHLDCNT ≠ 0) or
                     SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
    ELSE
        TMP_RDINFO.STATUS.CHILDPRESENT := (SECS(DS:RCX).CHLDCNT ≠ 0);
        TMP_RDINFO.STATUS.VIRTCHILDPRESENT :=
                    (SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
        TMP_RDINFO.ENCLAVECONTEXT := SECS(DS_RCX).ENCLAVECONTEXT;
    FI;
FI;

RAX := 0;
RFLAGS.ZF := 0;
RFLAGS.CF := 0;

DONE:
(* clear flags *)
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := ?0;
```

Flags Affected

ZF is set if ERDINFO fails due to concurrent operation with another SGX instruction; otherwise cleared.

CF is set if page is not a valid EPC page or not an EPC page; otherwise cleared.

PF, AF, OF, and SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

EREMOVE—Remove a page from the EPC

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLS[EREMOVE]	IR	V/V	SGX1	This leaf function removes a page from the EPC.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	EREMOVE (In)	Return error code (Out)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be un-associated with its SECS and be marked as unused. This instruction leaf can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if the operand is not properly aligned or does not refer to an EPC page or the page is in use by another thread, or other threads are running in the enclave to which the page belongs. In addition the instruction fails if the operand refers to an SECS with associations.

EREMOVE Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

The instruction faults if any of the following:

EREMOVE Faulting Conditions

The memory operand is not properly aligned.	The memory operand does not resolve in an EPC page.
Refers to an invalid SECS.	Refers to an EPC page that is locked by another thread.
Another Intel SGX instruction is accessing the EPC page.	RCX does not contain an effective address of an EPC page.
the EPC page refers to an SECS with associations.	

The error codes are:

Table 39-42. EREMOVE Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EREMOVE successful.
SGX_CHILD_PRESENT	If the SECS still have enclave pages loaded into EPC.
SGX_ENCLAVE_ACT	If there are still logical processors executing inside the enclave.

Concurrency Restrictions

Table 39-43. Base Concurrency Restrictions of EREMOVE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREMOVE	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Table 39-44. Additional Concurrency Restrictions of EREMOVE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREMOVE	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EREMOVE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve to an EPC page)
    THEN #PF(DS:RCX); FI;

TMP_SECS := Get_SECS_ADDRESS();

(* Check the EPC page for concurrency *)
IF (EPC page being referenced by another Intel SGX instruction)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address := DS:RCX;
            Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

(* if DS:RCX is already unused, nothing to do*)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT = PT_TRIM AND EPCM(DS:RCX).MODIFIED = 0))
    THEN GOTO DONE;
FI;
```

```
IF ( (EPCM(DS:RCX).PT = PT_VA) OR
    ((EPCM(DS:RCX).PT = PT_TRIM) AND (EPCM(DS:RCX).MODIFIED = 0)) )
    THEN
        EPCM(DS:RCX).VALID := 0;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT = PT_SECS)
    THEN
        IF (DS:RCX has an EPC page associated with it)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_CHILD_PRESENT;
                GOTO ERROR_EXIT;
        FI;
        (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
        IF (<<in VMX non-root operation>> AND
        <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
        (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_CHILD_PRESENT
                GOTO ERROR_EXIT
        FI;
        EPCM(DS:RCX).VALID := 0;
        GOTO DONE;
FI;

IF (Other threads active using SECS)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_ENCLAVE_ACT;
        GOTO ERROR_EXIT;
FI;

IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
 (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        EPCM(DS:RCX).VALID := 0;
        GOTO DONE;
FI;

DONE:
RAX := 0;
RFLAGS.ZF := 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the page.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If the memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the page.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the memory operand is not an EPC page.

ETRACK—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0CH ENCLS[ETRACK]	IR	V/V	SGX1	This leaf function activates EBLOCK checks.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	ETRACK (In)	Return error code (Out)	Pointer to the SECS of the EPC page (In)

Description

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

ETRACK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 39-45. ETRACK Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	ETRACK successful.
SGX_PREV_TRK_INCMPL	All processors did not complete the previous shoot-down sequence.

Concurrency Restrictions

Table 39-46. Base Concurrency Restrictions of ETRACK

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRACK	SECS [DS:RCX]	Shared	#GP	

Table 39-47. Additional Concurrency Restrictions of ETRACK

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRACK	SECS [DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT

Operation

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := TRACKING_RESOURCE_CONFLICT;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
                VMCS.Guest-linear_address := 0;
            Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PT ≠ PT_SECS)
    THEN #PF(DS:RCX); FI;

(* All processors must have completed the previous tracking cycle*)
IF ( (DS:RCX).TRACKING ≠ 0) )
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := TRACKING_REFERENCE_CONFLICT;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
                VMCS.Guest-linear_address := 0;
            Deliver VMEXIT;
        FI;
    RFLAGS.ZF := 1;
        RAX := SGX_PREV_TRK_INCMPL;
        GOTO DONE;
    ELSE
        RAX := 0;
        RFLAGS.ZF := 0;
FI;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another thread is concurrently using the tracking facility on this SECS.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

ETRACKC—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 11H ENCLS[ETRACKC]	IR	V/V	EAX[6]	This leaf function activates EBLOCK checks.

Instruction Operand Encoding

Op/En	EAX		RCX	
IR	ETRACK (In)	Return error code (Out)	Address of the destination EPC page (In, EA)	Address of the SECS page (In, EA)

Description

The ETRACKC instruction is thread safe variant of ETRACK leaf and can be executed concurrently with other CPU threads operating on the same SECS.

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

ETRACKC Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 39-48. ETRACKC Return Value in RAX

Error Code	Value	Description
No Error	0	ETRACKC successful.
SGX_EPC_PAGE_CONFLICT	7	Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD	6	Target page is not a VALID EPC page.
SGX_PREV_TRK_INCMPL	17	All processors did not complete the previous tracking sequence.
SGX_TRACK_NOT_REQUIRED	27	Target page type does not require tracking.

Concurrency Restrictions

Table 39-49. Base Concurrency Restrictions of ETRACKC

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRACKC	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS implicit	Concurrent		

Table 39-50. Additional Concurrency Restrictions of ETRACKC

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRACKC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT

Operation

Temp Variables in ETRACKC Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

```
(* check alignment of EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
#GP(0); FI;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
#PF(DS:RCX, PFEC.SGX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page is being modified) THEN
    RFLAGS.ZF := 1;
    RFLAGS.CF := 0;
    RAX := SGX_EPC_PAGE_CONFLICT;
    goto DONE_POST_LOCK_RELEASE;
FI;

(* check to make sure the page is valid *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    RFLAGS.ZF := 1;
    RFLAGS.CF := 0;
    RAX := SGX_PG_INVLD;
    GOTO DONE;
FI;

(* find out the target SECS page *)
IF (EPCM(DS:RCX).PT is PT_REG or PT_TCS or PT_TRIM or PT_SS_FIRST or PT_SS_REST) THEN
    TMP_SECS := Obtain SECS through EPCM(DS:RCX).ENCLAVESECS;
ELSE IF (EPCM(DS:RCX).PT is PT_SECS) THEN
    TMP_SECS := Obtain SECS through (DS:RCX);
ELSE
    RFLAGS.ZF := 0;
    RFLAGS.CF := 1;
    RAX := SGX_TRACK_NOT_REQUIRED;
    GOTO DONE;
FI;
```

```
(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS) THEN
    IF ((VMX non-root mode) and
    (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
        VMCS.Exit_reason := SGX_CONFLICT;
        VMCS.Exit_qualification.code := TRACKING_RESOURCE_CONFLICT;
        VMCS.Exit_qualification.error := 0;
        VMCS.Guest-physical_address :=
            SECS(TMP_SECS).ENCLAVECONTEXT;
        VMCS.Guest-linear_address := 0;
        Deliver VMEXIT;
    FI;

  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_EPC_PAGE_CONFLICT;
  GOTO DONE;
FI;
(* All processors must have completed the previous tracking cycle*)
IF ( (TMP_SECS).TRACKING ≠ 0) )
THEN
    IF ((VMX non-root mode) and
    (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
        VMCS.Exit_reason := SGX_CONFLICT;
        VMCS.Exit_qualification.code := TRACKING_REFERENCE_CONFLICT;
        VMCS.Exit_qualification.error := 0;
        VMCS.Guest-physical_address :=
            SECS(TMP_SECS).ENCLAVECONTEXT;
        VMCS.Guest-linear_address := 0;
        Deliver VMEXIT;
    FI;

  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_PREV_TRK_INCMPL;
  GOTO DONE;
FI;

RFLAGS.ZF := 0;
RFLAGS.CF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.PF,AF,OF,SF := 0;
```

Flags Affected

ZF is set if ETRACKC fails due to concurrent operations with another SGX instructions or target page is an invalid EPC page or tracking is not completed on SECS page; otherwise cleared.

CF is set if target page is not of a type that requires tracking; otherwise cleared.

PF, AF, OF, and SF are cleared.

Protected Mode Exceptions

#GP(0)	If the memory operand violates access-control policies of DS segment.
	If DS segment is unusable.
	If the memory operand is not properly aligned.
#PF(error code)	If the memory operand expected to be in EPC does not resolve to an EPC page.
	If a page fault occurs in access memory operand.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
#PF(error code)	If the memory operand expected to be in EPC does not resolve to an EPC page.
	If a page fault occurs in access memory operand.

EUPDATESVN—Update CR_CPUSVN

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 18H ENCLS[EUPDATESVN]	None	V/V	Bit 10	This leaf function updates the CR_CPUSVN if microcode has been updated and EPC is ready.

Description

If EPC is ready for SVN update, this leaf function updates CR_CPUSVN to the currently loaded microcode update SVN and generates new cryptographic assets. The EPC is ready when no page in the EPC is valid. EREMOVE should be used to mark all pages as unused.

It is the responsibility of system software to ensure that no other thread is executing or attempts to execute any ENCLS leaf while executing EUPDATESVN. Concurrency violations between EUPDATESVN and some ENCLS leaves may cause the ENCLS leaf to generate #GP(0) in ways unexpected to legacy software. System software should also prevent unnecessary software from having access to EUPDATESVN. For example, enable ENCLS exiting should be used to prevent VMs that are not part of the management system software from using EUPDATESVN.

The EUPDATESVN leaf function fails if an ENCLS instruction is in progress on any thread, the EPC is not ready for an update, or there is insufficient entropy in the random number generator. The ZF flag will be set to indicate an error and a code returned in RAX. If EUPDATESVN was successful but CR_CPUSVN was already up to date, the CF flag will be set and RAX will indicate that no update occurred.

If insufficient entropy causes a failure, software should repeat the instruction.

The error codes are:

Table 39-51. EUPDATESVN Return Value in RAX

Error Code (see Table 39-4)	Value	Description
No Error	0	EUPDATESVN successful.
SGX_EPC_PAGE_CONFLICT	7	An instruction concurrency rule was violated.
SGX_INSUFFICIENT_ENTROPY	29	RNG contains insufficient entropy.
SGX_EPC_NOT_READY	30	EPC is not ready for SVN update.
SGX_NO_UPDATE	31	EUPDATESVN was successful, but CR_CPUSVN was not updated because the current SVN is older than CR_CPUSVN.

Concurrency Restrictions

Table 39-52. Base Concurrency Restrictions of EUPDATESVN

Leaf	Base Concurrency Restrictions	
	Access	On Conflict
EUPDATESVN	Exclusive	SGX_EPC_PAGE_CONFLICT

Table 39-53. Additional Concurrency Restrictions of EUPDATESVN

Leaf	Additional Concurrency Restrictions	
	vs. EADD, EAUG, ECREATE, ELDB, ELDBC, ELDUC, EPA, EREMOVE, EWB	
	Access	On Conflict
EUPDATESVN	Exclusive	SGX_EPC_PAGE_CONFLICT

Operation

Temp Variables in EUPDATESVN Operational Flow

Name	Type	Size (Bytes)	Description
TMP_CPUSVN	CR_CPUSVN	16	Temporary copy of CR_CPUSVN prior to update.
TMP_KEY	Key	64	Temporary copy of new paging key.

```
IF (Other instruction is accessing EPC) THEN
    RFLAGS.ZF := 1
    RAX := SGX_EPC_PAGE_CONFLICT;
    GOTO ERROR_EXIT;
FI

(* Verify EPC is ready *)
IF (the EPC contains any valid pages) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_EPC_NOT_READY;
    GOTO ERROR_EXIT;
FI

(* Refresh paging key *)
TMP_KEY = (* Generate a 512-bit cryptographically random number *)
IF (insufficient entropy available) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INSUFFICIENT_ENTROPY;
    GOTO ERROR_EXIT;
FI

(* Commit *)
TMP_CPUSVN := CR_CPUSVN;

(* Update CR_CPUSVN to reflect current microcode update SVN *)

(* Determine if info status is needed *)
IF (TMP_CPUSVN = CR_CPUSVN) THEN
    RFLAGS.CF := 1;
    RAX := SGX_NO_UPDATE;
ELSE
    THEN
        CR_BASE_KEY := TMP_KEY[255:0];
        CR_REPORT_MAC_KEY := TMP_KEY[512:256];
FI
ERROR_EXIT:
```

Flags Affected

ZF is set if an error occurs; otherwise, cleared.

CF is set when the instruction is completed successfully and no SVN update was needed.

PF, AF, OF, and SF are cleared.

EWB—Invalidate an EPC Page and Write out to Main Memory

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0BH ENCLS[EWB]	IR	V/V	SGX1	This leaf function invalidates an EPC page and writes it out to main memory.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EWB (In)	Error code (Out)	Address of an PAGEINFO (In)	Address of the EPC page (In)	Address of a VA slot (In)

Description

This leaf function copies a page from the EPC to regular main memory. As part of the copying process, the page is cryptographically protected. This instruction can only be executed when current privilege level is 0.

The table below provides additional information on the memory parameter of EPA leaf function.

EWB Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	EPCPAGE	VASLOT
Non-EPC R/W access	Non-EPC R/W access	Non-EPC R/W access	EPC R/W access	EPC R/W access

The error codes are:

Table 39-54. EWB Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EWB successful.
SGX_PAGE_NOT_BLOCKED	If page is not marked as blocked.
SGX_NOT_TRACKED	If EWB is racing with ETRACK instruction.
SGX_VA_SLOT_OCCUPIED	Version array slot contained valid entry.
SGX_CHILD_PRESENT	Child page present while attempting to page out enclave.

Concurrency Restrictions

Table 39-55. Base Concurrency Restrictions of EWB

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EWB	Source [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	

Table 39-56. Additional Concurrency Restrictions of EWB

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EWB	Source [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Exclusive	

Operation

Temp Variables in EWB Operational Flow

Name	Type	Size (Bytes)	Description
TMP_SRCPGE	Memory page	4096	
TMP_PCMD	PCMD	128	
TMP_SECS	SECS	4096	
TMP_BPEPOCH	UINT64	8	
TMP_BPREFCOUNT	UINT64	8	
TMP_HEADER	MAC Header	128	
TMP_PCMD_ENCLAVEID	UINT64	8	
TMP_VER	UINT64	8	
TMP_PK	UINT128	16	

```
IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (DS:RDX is not 8Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

(* EPCPAGE and VASLOT should not resolve to the same EPC page*)
IF (DS:RCX and DS:RDX resolve to the same EPC page)
    THEN #GP(0); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
(* Note PAGEINFO.PCMD is overlaid on top of PAGEINFO.SECINFO *)
TMP_PCMD := DS:RBX.PCMD;

If (DS:RBX.LINADDR ≠ 0) OR (DS:RBX.SECS ≠ 0)
    THEN #GP(0); FI;

IF ( (DS:TMP_PCMD is not 128Byte Aligned) or (DS:TMP_SRCPGE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Check for concurrent Intel SGX instruction access to the page *)
IF (Other Intel SGX instruction is accessing page)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address := DS:RCX;
            Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

(*Check if the VA Page is being removed or changed*)
IF (VA Page is being modified)
    THEN #GP(0); FI;

(* Verify that EPCPAGE and VASLOT page are valid EPC pages and DS:RDX is VA *)
IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~FFFH).PT is not PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Perform page-type-specific exception checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM ) or
 (EPCM(DS:RCX).PT is PT_SS_FIRST ) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        TMP_SECS = Obtain SECS through EPCM(DS:RCX)
    (* Check that EBLOCK has occurred correctly *)
    IF (EBLOCK is not correct)
        THEN #GP(0); FI;
FI;

RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;

(* Zero out TMP_HEADER*)
TMP_HEADER[ sizeof(TMP_HEADER) - 1 : 0] := 0;

(* Perform page-type-specific checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM )or
 (EPCM(DS:RCX).PT is PT_SS_FIRST ) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        (* check to see if the page is evictable *)
        IF (EPCM(DS:RCX).BLOCKED = 0)
            THEN
                RAX := SGX_PAGE NOT_BLOCKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
        FI;
        (* Check if tracking done correctly *)
        IF (Tracking not correct)
            THEN
                RAX := SGX_NOT_TRACKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
        FI;

        (* Obtain EID to establish cryptographic binding between the paged-out page and the enclave *)
        TMP_HEADER.EID := TMP_SECS.EID;

        (* Obtain EID as an enclave handle for software *)
        TMP_PCMD_ENCLAVEID := TMP_SECS.EID;
    ELSE IF (EPCM(DS:RCX).PT is PT_SECS)
        (*check that there are no child pages inside the enclave *)
        IF (DS:RCX has an EPC page associated with it)
            THEN
                RAX := SGX_CHILD_PRESENT;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
        FI:
        (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
        IF (<<in VMX non-root operation>> AND
     <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
     (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_CHILD_PRESENT;
                GOTO ERROR_EXIT;
        FI;
        TMP_HEADER.EID := 0;
        (* Obtain EID as an enclave handle for software *)
        TMP_PCMD_ENCLAVEID := (DS:RCX).EID;
    ELSE IF (EPCM(DS:RCX).PT is PT_VA)
        TMP_HEADER.EID := 0; // Zero is not a special value
        (* No enclave handle for VA pages*)
        TMP_PCMD_ENCLAVEID := 0;
FI;

TMP_HEADER.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;
TMP_HEADER.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
TMP_HEADER.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
TMP_HEADER.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_HEADER.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_HEADER.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;

(* Encrypt the page, DS:RCX could be encrypted in place. AES-GCM produces 2 values, {ciphertext, MAC}. *)
(* AES-GCM input parameters: key, GCM Counter, MAC_HDR, MAC_HDR_SIZE, SRC, SRC_SIZE)*)
{DS:TMP_SRCPGE, DS:TMP_PCMD.MAC} := AES_GCM_ENC(CR_BASE_PK), (TMP_VER << 32),
    TMP_HEADER, 128, DS:RCX, 4096);

(* Write the output *)
Zero out DS:TMP_PCMD.SECINFO
DS:TMP_PCMD.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
DS:TMP_PCMD.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
DS:TMP_PCMD.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
DS:TMP_PCMD.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
DS:TMP_PCMD.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;
DS:TMP_PCMD.RESERVED := 0;
DS:TMP_PCMD.ENCLAVEID := TMP_PCMD_ENCLAVEID;
DS:RBX.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;

(*Check if version array slot was empty *)
```

```
IF ([DS.RDX])
    THEN
         RAX := SGX_VA_SLOT_OCCUPIED
         RFLAGS.CF := 1;
FI;

(* Write version to Version Array slot *)
[DS.RDX] := TMP_VER;

(* Free up EPCM Entry *)
EPCM.(DS:RCX).VALID := 0;
ERROR_EXIT:
```

Flags Affected

ZF is set if page is not blocked, not tracked, or a child is present. Otherwise cleared.

CF is set if VA slot is previously occupied, Otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the EPC page and VASLOT resolve to the same EPC page.
	If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
	If the tracking resource is in use.
	If the EPC page or the version array page is invalid.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If one of the EPC memory operands has incorrect page type.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the EPC page and VASLOT resolve to the same EPC page.
	If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
	If the tracking resource is in use.
	If the EPC page or the version array page in invalid.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If one of the EPC memory operands has incorrect page type.

39.4 INTEL® SGX USER LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLU instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of the implicitly-encoded register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EACCEPT—Accept Changes to an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLU[EACCEPT]	IR	V/V	SGX2	This leaf function accepts changes made by system software to an EPC page in the running enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EACCEPT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function accepts changes to a page in the running enclave by verifying that the security attributes specified in the SECINFO match the security attributes of the page in the EPCM. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPT leaf function.

EACCEPT Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EACCEPT Faulting Conditions

The operands are not properly aligned.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	Page type is PT_REG and MODIFIED bit is 0.
SECINFO contains an invalid request.	Page type is PT_TCS or PT_TRIM and PENDING bit is 0 and MODIFIED bit is 1.
If security attributes of the SECINFO page make the page inaccessible.	

The error codes are:

Table 39-57. EACCEPT Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EACCEPT successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.
SGX_NOT_TRACKED	The OS did not complete an ETRACK on the target page.

Concurrency Restrictions

Table 39-58. Base Concurrency Restrictions of EACCEPT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target [DS:RCX]	Shared	#GP	
	SECINFO [DS:RBX]	Concurrent		

Table 39-59. Additional Concurrency Restrictions of EACCEPT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EACCEPT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operands belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX is not within CR_ELRANGE)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
    (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or
    (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ (DS:RBX & FFFH)) )
    THEN #PF(DS:RBX); FI;

(* Copy 64 bytes of contents *)
SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero )
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;
```

```
IF (DS:RCX is not within CR_ELRANGE)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check that the combination of requested PT, PENDING, and MODIFIED is legal *)
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 0 )
    THEN
         IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) and
          ((SCRATCH_SECINFO.FLAGS.PR is 1) or
          (SCRATCH_SECINFO.FLAGS.PENDING is 1)) and
          (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) or
          ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS or PT_TRIM) and
          (SCRATCH_SECINFO.FLAGS.PR is 0) and
          (SCRATCH_SECINFO.FLAGS.PENDING is 0) and
          (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) )))
               THEN #GP(0); FI
    ELSE
         IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) AND
          ((SCRATCH_SECINFO.FLAGS.PR is 1) OR
          (SCRATCH_SECINFO.FLAGS.PENDING is 1)) AND
          (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) OR
          ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS OR PT_TRIM) AND
          (SCRATCH_SECINFO.FLAGS.PENDING is 0) AND
          (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) AND
          (SCRATCH_SECINFO.FLAGS.PR is 0)) OR
          ((SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST or PT_SS_REST) AND
          (SCRATCH_SECINFO.FLAGS.PENDING is 1) AND
          (SCRATCH_SECINFO.FLAGS.MODIFIED is 0) AND
          (SCRATCH_SECINFO.FLAGS.PR is 0))))
               THEN #GP(0); FI;
    FI;

(* Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).BLOCKED is not 0) or
  ((EPCM(DS:RCX).PT is not PT_REG) and (EPCM(DS:RCX).PT is not PT_TCS) and (EPCM(DS:RCX).PT is not PT_TRIM)
  and (EPCM(DS:RCX).PT is not PT_SS_FIRST) and (EPCM(DS:RCX).PT is not PT_SS_REST)) or
  (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS))
    THEN #PF((DS:RCX); FI;

(* Check the destination EPC page for concurrency *)
IF ( EPC page in use )
    THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN #PF(DS:RCX); FI;

(* Verify that accept request matches current EPC page settings *)
IF ( (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX) or (EPCM(DS:RCX).PENDING ≠ SCRATCH_SECINFO.FLAGS.PENDING) or
    (EPCM(DS:RCX).MODIFIED ≠ SCRATCH_SECINFO.FLAGS.MODIFIED) or (EPCM(DS:RCX).R ≠ SCRATCH_SECINFO.FLAGS.R) or
    (EPCM(DS:RCX).W ≠ SCRATCH_SECINFO.FLAGS.W) or (EPCM(DS:RCX).X ≠ SCRATCH_SECINFO.FLAGS.X) or
    (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;
(* Check that all required threads have left enclave *)
IF (Tracking not correct)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_NOT_TRACKED;
        GOTO DONE;
FI;

(* Get pointer to the SECS to which the EPC page belongs *)
TMP_SECS = << Obtain physical address of SECS through EPCM(DS:RCX)>>
(* For TCS pages, perform additional checks *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
    THEN
        IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;

    (* Check that TCS.FLAGS.DBGOPTIN, TCS stack, and TCS status are correctly initialized *)
    (* check that TCS.PREVSSP is 0 *)
    IF ( ((DS:RCX).FLAGS.DBGOPTIN is not 0) or ((DS:RCX).CSSA ≥ (DS:RCX).NSSA) or ((DS:RCX).AEP is not 0) or ((DS:RCX).STATE is not 0)
or ((CPUID.(EAX=07H, ECX=0H):ECX[CET_SS] = 1) AND ((DS:RCX).PREVSSP != 0)))
        THEN #GP(0); FI;

    (* Check consistency of FS & GS Limit *)
    IF ( (TMP_SECS.ATTRIBUTES.MODE64BIT is 0) and ((DS:RCX.FSLIMIT & FFFH ≠ FFFH) or (DS:RCX.GSLIMIT & FFFH ≠ FFFH)) )
        THEN #GP(0); FI;
FI;

(* Clear PENDING/MODIFIED flags to mark accept operation complete *)
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;

(* Clear EAX and ZF to indicate successful completion *)
RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if page cannot be accepted, otherwise cleared. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

EACCEPTCOPY—Initialize a Pending Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLU[EACCEPTCOPY]	IR	V/V	SGX2	This leaf function initializes a dynamically allocated EPC page from another page in the EPC.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EACCEPTCOPY (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)	Address of the source EPC page (In)

Description

This leaf function copies the contents of an existing EPC page into an uninitialized EPC page (created by EAUG). After initialization, the instruction may also modify the access rights associated with the destination EPC page. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX and RDX each contain the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPTCOPY leaf function.

EACCEPTCOPY Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)	EPCPAGE (Source)
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EACCEPTCOPY Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	If security attributes of the source EPC page make the page inaccessible.
The EPC page is not valid.	RBX does not contain an effective address in an EPC page in the running enclave.
SECINFO contains an invalid request.	RCX/RDX does not contain an effective address of an EPC page in the running enclave.

The error codes are:

Table 39-60. EACCEPTCOPY Return Value in RAX

Error Code (see Table 39-4)	Description
No Error	EACCEPTCOPY successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.

Concurrency Restrictions

Table 39-61. Base Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPTCOPY	Target [DS:RCX]	Concurrent		
	Source [DS:RDX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

Table 39-62. Additional Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPTCOPY	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EACCEPTCOPY Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF ( (DS:RCX is not 4KByte Aligned) or (DS:RDX is not 4KByte Aligned) )
    THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) or (DS:RDX is not within CR_ELRANGE))
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
    (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or
    (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ DS:RBX) )
    THEN #PF(DS:RBX); FI;
```

```
(* Copy 64 bytes of contents *)
SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or (SCRATCH_SECINFO.FLAGS.R=0) AND(SCRATCH_SECINFO.FLAGS.W≠0 ) or
    (SCRATCH_SECINFO.FLAGS.PT is not PT_REG) )
    THEN #GP(0); FI;

(* Check security attributes of the source EPC page *)
IF ( (EPCM(DS:RDX).VALID = 0) or (EPCM(DS:RCX).R = 0) or (EPCM(DS:RDX).PENDING ≠ 0) or (EPCM(DS:RDX).MODIFIED ≠ 0) or
    (EPCM(DS:RDX).BLOCKED ≠ 0) or (EPCM(DS:RDX).PT ≠ PT_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RDX).ENCLAVEADDRESS ≠ DS:RDX))
    THEN #PF(DS:RDX); FI;

(* Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 1) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RDX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;

(* Check the destination EPC page for concurrency *)
IF (destination EPC page in use )
    THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 1) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).R ≠ 1) or (EPCM(DS:RCX).W ≠ 1) or (EPCM(DS:RCX).X ≠ 0) or
    (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;

(* Copy 4KBbytes form the source to destination EPC page*)
DS:RCX[32767:0] := DS:RDX[32767:0];

(* Update EPCM permissions *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PENDING := 0;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

Sets ZF if page is not modifiable, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

EDECCSSA—Decrements TCS.CSSA

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 09H ENCLU[EDECCSSA]	IR	V/V	EDECCSSA	This leaf function decrements TCS.CSSA.

Instruction Operand Encoding

Op/En	EAX
IR	EDECCSSA (In)

Description

This leaf function changes the current SSA frame by decrementing TCS.CSSA for the current enclave thread. If the enclave has enabled CET shadow stacks or indirect branch tracking, then EDECCSSA also changes the current CET state save frame. This instruction leaf can only be executed inside an enclave.

EDECCSSA Memory Parameter Semantics

TCS
Read/Write access by Enclave

The instruction faults if any of the following:

EDECCSSA Faulting Conditions

TCS.CSSA is 0.	TCS is not valid or available or locked.
The SSA frame is not valid or in use.	

Concurrency Restrictions

Table 39-63. Base Concurrency Restrictions of EDECCSSA

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDECCSSA	TCS [CR_TCS_PA]	Shared	#GP	

Table 39-64. Additional Concurrency Restrictions of EDECCSSA

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDECCSSA	TCS [CR_TCS_PA]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDECCSSA Operational Flow

Name	Type	Size (bits)	Description
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	Integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the target frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the target SSA frame.
TMP_XSAVE_PAGE_PA_n	Physical Address	32/64	Physical address of the nth page within the target SSA frame.
TMP_CET_SAVE_AREA	Effective Address	32/64	Address of the current CET save area.
TMP_CET_SAVE_PAGE	Effective Address	32/64	Address of the current CET save area page.

```
(* Check concurrency of TCS operation *)
IF (Other Intel SGX instructions are operating on TCS)
    THEN #GP(0); FI;

IF (CR_TCS_PA.CSSA = 0)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := CR_TCS_PA.OSSA + CR_ACTIVE_SECS.BASEADDR + 4096 * CR_ACTIVE_SECS.SSAFRAMESIZE * (CR_TCS_PA.CSSA - 1);
TMP_XSIZE := compute_XSAVE_frame_size(CR_ACTIVE_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE_.MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or
    (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
    (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS) or
    (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    TMP_XSAVE_PAGE_PA_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * CR_ACTIVE_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
```

```
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or
    (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
        THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (sizeof(GPRSGX_AREA) -1) is not in DS segment)
            THEN #GP(0); FI;
FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        IF ((CR_ACTIVE_SECS.CET_ATTRIBUTES.SH_STK_EN == 1) OR (CR_ACTIVE_SECS.CET_ATTRIBUTES.ENDBR_EN == 1))
            THEN
                (* Compute linear address of what will become new CET state save area and cache its PA *)
                TMP_CET_SAVE_AREA := CR_TCS_PA.OCETSSA + CR_ACTIVE_SECS.BASEADDR + (CR_TCS_PA.CSSA - 1) * 16;
                TMP_CET_SAVE_PAGE := TMP_CET_SAVE_AREA & ~0xFFF;
                Check the TMP_CET_SAVE_PAGE page is read/write accessible
                If fault occurs release locks, abort and deliver fault

                (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
                IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS))
            THEN #PF(DS:TMP_CET_SAVE_PAGE); FI;
        FI;
FI;

(* At this point, the instruction is guaranteed to complete *)
CR_TCS_PA.CSSA := CR_TCS_PA.CSSA - 1;

CR_GPR_PA := Physical_Address(DS:TMP_GPR);

FOR EACH TMP_XSAVE_PAGE_n
    CR_XSAVE_PAGE_n := TMP_XSAVE_PAGE_PA_n;
```

```
ENDFOR

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        IF ((TMP_SECS.CET_ATTRIBUTES.SH_STK_EN == 1) OR
        (TMP_SECS.CET_ATTRIBUTES.ENDBR_EN == 1))
            THEN
                CR_CET_SAVE_AREA_PA := Physical_Address(DS:TMP_CET_SAVE_AREA);
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If CR_TCS_PA.CSSA = 0.
#PF(error code)	If a page fault occurs in accessing memory.
	If one or more pages of the target SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.
	If CET is enabled for the enclave and the target CET SSA frame is not readable/writable, or does not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If CR_TCS_PA.CSSA = 0.
#PF(error code)	If a page fault occurs in accessing memory.
	If one or more pages of the target SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.
	If CET is enabled for the enclave and the target CET SSA frame is not readable/writable, or does not resolve to a valid PT_REG EPC page.

EENTER—Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLU[EENTER]	IR	V/V	SGX1	This leaf function is used to enter an enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	
IR	EENTER (In)	Content of RBX.CSSA (Out)	Address of a TCS (In)	Address of AEP (In)	Address of IP following EENTER (Out)

Description

The ENCLU[EENTER] instruction transfers execution to an enclave. At the end of the instruction, the logical processor is executing in enclave mode at the RIP computed as EnclaveBase + TCS.OENTRY. If the target address is not within the CS segment (32-bit) or is not canonical (64-bit), a #GP(0) results.

EENTER Memory Parameter Semantics

TCS
Enclave access

EENTER is a serializing instruction. The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use.	Either of TCS-specified FS and GS segment is not a subsets of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.	

The following operations are performed by EENTER:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or interrupt.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 41.1.2):
  - — On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 41.2.5).
  - — On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 41.2.2).

- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 41.2.3):
  - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
  - PEBS is suppressed.
  - AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set
  - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Table 39-65. Base Concurrency Restrictions of EENTER

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EENTER	TCS [DS:RBX]	Shared	#GP	

Table 39-66. Additional Concurrency Restrictions of EENTER

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EENTER	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EENTER Operational Flow

Name	Type	Size (Bits)	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or DS[bits 11:9] != 001B))
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
    THEN
```

```
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
        IF(ES usable and ES.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions are operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS) )
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Ensure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFFFFFFFFFCH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor’s operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;
```

```
IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUES.XFRM) THEN #GP(0); FI;
FI;

IF ((DS:RBX).CSSA.FLAGS.DBGOPTIN = 0) and (DS:RBX).CSSA.FLAGS.AEXNOTIFY ≠ TMP_SECS.ATTRIBUTES.AEXNOTIFY))
    THEN #GP(0); FI;

(* Make sure the SSA contains at least one more frame *)
IF ( (DS:RBX).CSSA ≥ (DS:RBX).NSSA)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * (DS:RBX).CSSA;
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort, and deliver that fault;

    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
If a fault occurs; release locks, abort, and deliver that fault;

IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
```

```
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA := Physical_Address (DS: TMP_GPR);

(* Validate TCS.OENTRY *)
TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
            THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE)
    THEN #GP(0); FI;

TMP_IA32_U_CET := 0
```

```
TMP_SSP : = 0

IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
    THEN
        IF ( CR4.CET = 0 )
            THEN
                (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
                IF ( TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0 ) #GP(0); FI;
        FI;
        (* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail EENTER *)
        IF ( TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1 )
            THEN
                IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;
        FI;

    IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
        THEN
            (* Setup CET state from SECS, note tracker goes to IDLE *)
            TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
            IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1 )
                THEN
                    TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;
                    TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
            FI;

            (* Compute linear address of what will become new CET state save area and cache its PA *)
            TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16
            TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

            Check the TMP_CET_SAVE_PAGE page is read/write accessible
            If fault occurs release locks, abort, and deliver fault

            (* Read the EPCM VALID, PENDING, MODIFIED, BLOCKED, and PT fields atomically *)
            IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
             (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
                THEN
                    #PF(DS:TMP_CET_SAVE_PAGE);
            FI;

            CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)

            IF TMP_IA32_U_CET.SH_STK_EN = 1
                THEN
                    TMP_SSP = TCS.PREVSSP;
            FI;
    FI;
```

```
FI;

CR_ENCLAVE_MODE := 1;
CR_ACTIVE_SECS := TMP_SECS;
CR_ELRANGE := (TMPSECS.BASEADDR, TMP_SECS.SIZE);

(* Save state for possible AEXs *)
CR_TCS_PA := Physical_Address (DS:RBX);
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 := XCR0;
    XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
FI;

RCX := RIP;
RIP := TMP_TARGET;
RAX := (DS:RBX).CSSA;
(* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
DS:TMP_SSA.U_RSP := RSP;
DS:TMP_SSA.U_RBP := RBP;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS[bit 9];
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS[bit 21];
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS[bit 9];
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS[bit 21];
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        CR_SAVE_TF := RFLAGS.TF;
        RFLAGS.TF := 0;
        Suppress_monitor_trap_flag for the source of the execution of the enclave;
        Suppress any pending debug exceptions;
        Suppress any pending MTF VM exit;
    ELSE
        IF RFLAGS.TF = 1
            THEN pend a single-step #DB at the end of EENTER; FI;
        IF the “monitor trap flag” VM-execution control is set
            THEN pend an MTF VM exit at the end of EENTER; FI;
FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7H, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET := IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP := SSP
                SSP := TMP_SSP
        FI;

        IA32_U_CET := TMP_IA32_U_CET;

FI;

Flush_linear_context;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry.

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If any reserved field in the TCS FLAG is set.
	If the target address is not within the CS segment.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.
#PF(error code)	If a page fault occurs in accessing memory.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the target address is not canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

EEXIT—Exits an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLU[EEXIT]	IR	V/V	SGX1	This leaf function is used to exit an enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EEXIT (In)	Target address outside the enclave (In)	Address of the current AEP (Out)

Description

The ENCLU[EEXIT] instruction exits the currently executing enclave and branches to the location specified in RBX. RCX receives the current AEP. If RBX is not within the CS (32-bit mode) or is not canonical (64-bit mode) a #GP(0) results.

EEXIT Memory Parameter Semantics

Target Address
Non-Enclave read and execute access

If RBX specifies an address that is inside the enclave, the instruction will complete normally. The fetch of the next instruction will occur in non-enclave mode, but will attempt to fetch from inside the enclave. This fetch returns a fixed data pattern.

If secrets are contained in any registers, it is responsibility of enclave software to clear those registers.

If XCR0 was modified on enclave entry, it is restored to the value it had at the time of the most recent EENTER or ERESUME.

If the enclave is opt-out, RFLAGS.TF is loaded from the value previously saved on EENTER.

Code and data breakpoints are unsuppressed.

Performance monitoring counters are unsuppressed.

Concurrency Restrictions

Table 39-67. Base Concurrency Restrictions of EEXIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXIT		Concurrent		

Table 39-68. Additional Concurrency Restrictions of EEXIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXIT		Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EEXIT Operational Flow

Name	Type	Size (Bits)	Description
TMP_RIP	Effective Address	32/64	Saved copy of CRIP for use when creating LBR.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (TMP_MODE64 = 1)
    THEN
        IF (RBX is not canonical) THEN #GP(0); FI;
    ELSE
        IF (RBX > CS limit) THEN #GP(0); FI;
FI;

TMP_RIP := CRIP;
RIP := RBX;

(* Return current AEP in RCX *)
RCX := CR_TCS_PA.AEP;

(* Do the FS/GS swap *)
FS.selector := CR_SAVE_FS.selector;
FS.base := CR_SAVE_FS.base;
FS.limit := CR_SAVE_FS.limit;
FS.access_rights := CR_SAVE_FS.access_rights;
GS.selector := CR_SAVE_GS.selector;
GS.base := CR_SAVE_GS.base;
GS.limit := CR_SAVE_GS.limit;
GS.access_rights := CR_SAVE_GS.access_rights;

(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
    XCR0 := CR_SAVE__XCR0;
FI;

Unsuppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        UnSuppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        Restore suppressed breakpoint matches;
        RFLAGS.TF := CR_SAVE_TF;
        UnSuppress_montior_trap_flag;
        UnSuppress_LBR_Generation;
        UnSuppress_performance monitoring_activity;
        Restore performance monitoring counter AnyThread demotion to MyThread in enclave back to AnyThread
FI;

IF RFLAGS.TF = 1
    THEN Pend Single-Step #DB at the end of EEXIT;
FI;
```

```
IF the "monitor trap flag" VM-execution control is set
    THEN pend a MTF VM exit at the end of EEXIT;
FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        (* Record PREVSSP *)
        IF (IA32_U_CET.SH_STK_EN == 1)
            THEN CR_TCS_PA.PREVSSP = SSP; FI;
FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* Restore enclosing app's CET state from the save registers *)
        IA32_U_CET := CR_SAVE_IA32_U_CET;
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN SSP := CR_SAVE_SSP; FI;

        (* Update enclosing app's TRACKER if enclosing app has indirect branch tracking enabled *)
        IF (CR4.CET = 1 AND IA32_U_CET.ENDBR_EN = 1)
            THEN
                IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPPRESS := 0
        FI;
FI;

CR_ENCLAVE_MODE := 0;
CR_TCS_PA.STATE := INACTIVE;

(* Assure consistent translations *)
Flush_linear_context;
```

Flags Affected

RFLAGS.TF is restored from the value previously saved in EENTER or ERESUME.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RBX is outside the CS segment.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RBX is not canonical.
#PF(error code)	If a page fault occurs in accessing memory operands.

EGETKEY—Retrieves a Cryptographic Key

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLU[EGETKEY]	IR	V/V	SGX1	This leaf function retrieves a cryptographic key.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EGETKEY (In)	Return error code (Out)	Address to a KEYREQUEST (In)	Address of the OUTPUTDATA (In)

Description

The ENCLU[EGETKEY] instruction returns a 128-bit secret key from the processor specific key hierarchy. The register RBX contains the effective address of a KEYREQUEST structure, which the instruction interprets to determine the key being requested. The Requesting Keys section below provides a description of the keys that can be requested. The RCX register contains the effective address where the key will be returned. Both the addresses in RBX & RCX should be locations inside the enclave.

EGETKEY derives keys using a processor unique value to create a specific key based on a number of possible inputs. This instruction leaf can only be executed inside an enclave.

EEGETKEY Memory Parameter Semantics

KEYREQUEST	OUTPUTDATA
Enclave read access	Enclave write access

After validating the operands, the instruction determines which key is to be produced and performs the following actions:

- The instruction assembles the derivation data for the key based on the Table 39-69.
- Computes derived key using the derivation data and package specific value.
- Outputs the calculated key to the address in RCX.

The instruction fails with #GP(0) if the operands are not properly aligned. Successful completion of the instruction will clear RFLAGS.{ZF, CF, AF, OF, SF, PF}. The instruction returns an error code if the user tries to request a key based on an invalid CR_CPUSVN or ISVSVN (when the user request is accepted, see the table below), requests a key for which it has not been granted the attribute to request, or requests a key that is not supported by the hardware. These checks may be performed in any order. Thus, an indication by error number of one cause (for example, invalid attribute) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same Enclave. The correctness of software should not rely on the order resulting from the checks documented in this section. In such cases the ZF flag is set and the corresponding error bit (SGX_INVALID_SVN, SGX_INVALID_ATTRIBUTE, SGX_INVALID_KEYNAME) is set in RAX and the data at the address specified by RCX is unmodified.

Requesting Keys

The KEYREQUEST structure (see Section 36.18.1) identifies the key to be provided. The Keyrequest.KeyName field identifies which type of key is requested.

Deriving Keys

Key derivation is based on a combination of the enclave specific values (see Table 39-69) and a processor key. Depending on the key being requested a field may either be included by definition or the value may be included from the KeyRequest. A "yes" in Table 39-69 indicates the value for the field is included from its default location, identified in the source row, and a "request" indicates the values for the field is included from its corresponding KeyRequest field.

Table 39-69. Key Derivation

	Key Name	Attributes	Owner Epoch	CPU SVN	ISV SVN	ISV PRODID	ISVEXT PRODID	ISVFAM ILYID	MRENCLAVE	MRSIGNER	CONFIG ID	CONFIGS VN	RAND
Source	Key Dependent Constant	Y := SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;	CR_SGX OWNER EPOCH	Y := CPUSVN Register;	R := Req.ISV SVN;	SECS. ISVID	SECS.IS VEXTPR ODID	SECS.IS VFAMIL YID	SECS. MRENCLAVE	SECS. MRSIGNER	SECS.CO NFIGID	SECS.CO NFIGSVN	Req. KEYID
		R := AttribMask & SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;		R := Req.CPU SVN;									
EINITTOKEN	Yes	Request	Yes	Request	Request	Yes	No	No	No	Yes	No	No	Request
Report	Yes	Yes	Yes	Yes	No	No	No	No	Yes	No	Yes	Yes	Request
Seal	Yes	Request	Yes	Request	Request	Request	Request	Request	Request	Request	Request	Request	Request
Provisioning	Yes	Request	No	Request	Request	Yes	No	No	No	Yes	No	No	Yes
Provisioning Seal	Yes	Request	No	Request	Request	Request	Request	Request	No	Yes	Request	Request	Yes

Keys that permit the specification of a CPU or ISV's code's, or enclave configuration's SVNs have additional requirements. The caller may not request a key for an SVN beyond the current CPU, ISV or enclave configuration's SVN, respectively.

Several keys are access controlled. Access to the Provisioning Key and Provisioning Seal key requires the enclave's ATTRIBUTES.PROVISIONKEY be set. The EINITTOKEN Key requires ATTRIBUTES.EINITTOKEN_KEY be set and SECS.MRSIGNER equal IA32_SGXLEPUBKEYHASH.

Some keys are derived based on a hardcode PKCS padding constant (352 byte string):

HARDCODED_PKCS1_5_PADDING[15:0] := 0100H;

HARDCODED_PKCS1_5_PADDING[2655:16] := SignExtend330Byte(-1); // 330 bytes of 0FFH

HARDCODED_PKCS1_5_PADDING[2815:2656] := 2004000501020403650148866009060D30313000H;

The error codes are:

Table 39-70. EGETKEY Return Value in RAX

Error Code (see Table 39-4)	Value	Description
No Error	0	EGETKEY successful.
SGX_INVALID_ATTRIBUTE		The KEYREQUEST contains a KEYNAME for which the enclave is not authorized.
SGX_INVALID_CPUSVN		If KEYREQUEST.CPUSVN is an unsupported platforms CPUSVN value.
SGX_INVALID_ISVSVN		If KEYREQUEST software SVN (ISVSVN or CONFIGSVN) is greater than the enclave's corresponding SVN.
SGX_INVALID_KEYNAME		If KEYREQUEST.KEYNAME is an unsupported value.

Concurrency Restrictions

Table 39-71. Base Concurrency Restrictions of EGETKEY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		
	OUTPUTDATA [DS:RCX]	Concurrent		

Table 39-72. Additional Concurrency Restrictions of EGETKEY

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EGETKEY Operational Flow

Name	Type	Size (Bits)	Description
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_ATTRIBUTES		128	Temp Space for the calculation of the sealable Attributes.
TMP_ISVEXTPRODID		16 bytes	Temp Space for ISVEXTPRODID.
TMP_ISVPRODID		2 bytes	Temp Space for ISVPRODID.
TMP_ISVFAMILYID		16 bytes	Temp Space for ISVFAMILYID.
TMP_CONFIGID		64 bytes	Temp Space for CONFIGID.
TMP_CONFIGSVN		2 bytes	Temp Space for CONFIGSVN.
TMP_OUTPUTKEY		128	Temp Space for the calculation of the key.

```
(* Make sure KEYREQUEST is properly aligned and inside the current enclave *)
IF ( (DS:RBX is not 512Byte aligned) or (DS:RBX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

(* Make sure DS:RBX is an EPC address and the EPC page is valid *)
IF ( (DS:RBX does not resolve to an EPC address) or (EPCM(DS:RBX).VALID = 0) )
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
    (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )
    THEN #PF(DS:RBX);
FI;

(* Make sure OUTPUTDATA is properly aligned and inside the current enclave *)
IF ( (DS:RCX is not 16Byte aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

(* Make sure DS:RCX is an EPC address and the EPC page is valid *)
IF ( (DS:RCX does not resolve to an EPC address) or (EPCM(DS:RCX).VALID = 0) )
```

```
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
    THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).W = 0) )
    THEN #PF(DS:RCX);
FI;

(* Verify RESERVED spaces in KEYREQUEST are valid *)
IF ( (DS:RBX).RESERVED ≠ 0) or (DS:RBX.KEYPOLICY.RESERVED ≠ 0) )
    THEN #GP(0); FI;

TMP_CURRENTSECS := CR_ACTIVE_SECS;

(* Verify that CONFIGSVN & New Policy bits are not used if KSS is not enabled *)
IF ((TMP_CURRENTSECS.ATTRIBUTES.KSS == 0) AND ((DS:RBX.KEYPOLICY & 0x003C ≠ 0) OR (DS:RBX.CONFIGSVN > 0)))
    THEN #GP(0); FI;
(* Determine which enclave attributes that must be included in the key. Attributes that must always be include INIT & DEBUG *)
REQUIRED_SEALING_MASK[127:0] := 00000000 00000000 00000000 00000003H;
TMP_ATTRIBUTES := (DS:RBX.ATTRIBUTEMASK | REQUIRED_SEALING_MASK) & TMP_CURRENTSECS.ATTRIBUTES;

(* Compute MISCSELECT fields to be included *)
TMP_MISCSELECT := DS:RBX.MISCMASK & TMP_CURRENTSECS.MISCSELECT

(* Compute CET_ATTRIBUTES fields to be included *)
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN TMP_CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES_ MASK & TMP_CURRENTSECS.CET_ATTRIBUTES; FI;
TMP_KEYDEPENDENCIES := 0;

CASE (DS:RBX.KEYNAME)
    SEAL_KEY:
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_INVALID_CPUSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.CONFIGSVN > TMP_CURRENTSECS.CONFIGSVN)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;

        (*Include enclave identity?*)
```

```
            TMP_MRENCLAVE := 0;
            IF (DS:RBX.KEYPOLICY.MRENCLAVE = 1)
                THEN TMP_MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
            FI;
            (*Include enclave author?*)
            TMP_MRSIGNER := 0;
            IF (DS:RBX.KEYPOLICY.MRSIGNER = 1)
                THEN TMP_MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
            FI;
(* Include enclave product family ID? *)
    TMP_ISVFAMILYID := 0;
    IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
        THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
            FI;

    (* Include enclave product ID? *)
    TMP_ISVPRODID := 0;
    IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
        TMP_ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
            FI;

    (* Include enclave Config ID? *)
    TMP_CONFIGID := 0;
    TMP_CONFIGSVN := 0;
    IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
        TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;
        TMP_CONFIGSVN := DS:RBX.CONFIGSVN;
            FI;

    (* Include enclave extended product ID? *)
    TMP_ISVEXTPRODID := 0;
    IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1 )
        TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
    FI;

            //Determine values key is based on
            TMP_KEYDEPENDENCIES.KEYNAME := SEAL_KEY;
            TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
            TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
            TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
            TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
            TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
            TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
            TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTEMASK;
            TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_MRENCLAVE;
            TMP_KEYDEPENDENCIES.MRSIGNER := TMP_MRSIGNER;
            TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
            TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
            TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
            TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
            TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
            TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
            TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
            TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;
            TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
            IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
                THEN
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := DS:RBX.CET_ATTRIBUTES _MASK;
            FI;
            BREAK;
        REPORT_KEY:
            //Determine values key is based on
            TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
            TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
            TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
            TMP_KEYDEPENDENCIES.ISVPRODID := 0;
            TMP_KEYDEPENDENCIES.ISVSVN := 0;
            TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
            TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;
            TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
            TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
            TMP_KEYDEPENDENCIES.MRSIGNER := 0;
            TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
            TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
            TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
            TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
            TMP_KEYDEPENDENCIES.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
            TMP_KEYDEPENDENCIES.MISCMASK := 0;
            TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
            TMP_KEYDEPENDENCIES.CONFIGID := TMP_CURRENTSECS.CONFIGID;
            TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
            IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
                THEN
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES;
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
            FI;
            BREAK;
        EINITTOKEN_KEY:
            (* Check ENCLAVE has EINITTOKEN Key capability *)
            IF (TMP_CURRENTSECS.ATTRIBUTES.EINITTOKEN_KEY = 0)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_ATTRIBUTE;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.CPUSVN is beyond current CPU configuration)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_CPUSVN;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_ISVSVN;
                    GOTO EXIT;
            FI;
            (* Determine values key is based on *)
            TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
            TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
            TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
            TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID
            TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
            TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
            TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
            TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
            TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
            TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
            TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
            TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
            TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
            TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
            TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
            TMP_KEYDEPENDENCIES.MISCMASK := 0;
            TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
            TMP_KEYDEPENDENCIES.CONFIGID := 0;
            TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
            IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
                THEN
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := 0;
            FI;
            BREAK;
        PROVISION_KEY:
        (* Check ENCLAVE has PROVISIONING capability *)
            IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_ATTRIBUTE;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.CPUSVN is beyond current CPU configuration)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_CPUSVN;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_ISVSVN;
                    GOTO EXIT;
            FI;
            (* Determine values key is based on *)
            TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_KEY;
            TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
            TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
            TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
            TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
            TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
            TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
```

```
            TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTEMASK;
            TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
            TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
            TMP_KEYDEPENDENCIES.KEYID := 0;
            TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := 0;
            TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
            TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
            TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
            TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
            TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
            TMP_KEYDEPENDENCIES.CONFIGID := 0;
            IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
                THEN
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := 0;
            FI;
            BREAK;
        PROVISION_SEAL_KEY:
            (* Check ENCLAVE has PROVISIONING capability *)
            IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_ATTRIBUTE;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.CPUSVN is beyond current CPU configuration)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_CPUSVN;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
                THEN
                    RFLAGS.ZF := 1;
                    RAX := SGX_INVALID_ISVSVN;
                    GOTO EXIT;
            FI;
(* Include enclave product family ID? *)
    TMP_ISVFAMILYID := 0;
    IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
        THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
            FI;

    (* Include enclave product ID? *)
    TMP_ISVPRODID := 0;
    IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
        TMP_ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
            FI;

    (* Include enclave Config ID? *)
    TMP_CONFIGID := 0;
    TMP_CONFIGSVN := 0;
    IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
        TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;
        TMP_CONFIGSVN := DS:RBX.CONFIGSVN;
            FI;

    (* Include enclave extended product ID? *)
    TMP_ISVEXTPRODID := 0;
    IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1)
        TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
    FI;

            (* Determine values key is based on *)
            TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_SEAL_KEY;
            TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
            TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
            TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
            TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
            TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
            TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
            TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTEMASK;
            TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
            TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
            TMP_KEYDEPENDENCIES.KEYID := 0;
            TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
            TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
            TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
            TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
            TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
            TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
            TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;
            TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
            IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
                THEN
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := 0;
            FI;
            BREAK;
      DEFAULT:
            (* The value of KEYNAME is invalid *)
            RFLAGS.ZF := 1;
            RAX := SGX_INVALID_KEYNAME;
            GOTO EXIT:
ESAC;

(* Calculate the final derived key and output to the address in RCX *)
TMP_OUTPUTKEY := derivekey(TMP_KEYDEPENDENCIES);
DS:RCX[15:0] := TMP_OUTPUTKEY;
RAX := 0;
RFLAGS.ZF := 0;

EXIT:
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the current enclave.
	If an effective address is not properly aligned.
	If an effective address is outside the DS segment limit.
	If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the current enclave.
	If an effective address is not properly aligned.
	If an effective address is not canonical.
	If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory operands.

EMODPE—Extend an EPC Page Permissions

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLU[EMODPE]	IR	V/V	SGX2	This leaf function extends the access rights of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EMODPE (In)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function extends the access rights associated with an existing EPC page in the running enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not extend the page permissions will have no effect. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPE leaf function.

EMODPE Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EMODPE Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is not valid.	RCX does not contain an effective address of an EPC page in the running enclave.
SECINFO contains an invalid request.	

Concurrency Restrictions

Table 39-73. Base Concurrency Restrictions of EMODPE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPE	Target [DS:RCX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

Table 39-74. Additional Concurrency Restrictions of EMODPE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPE	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EMODPE Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RBX).VALID = 0) or (EPCM(DS:RBX).R = 0) or (EPCM(DS:RBX).PENDING ≠ 0) or (EPCM(DS:RBX).MODIFIED ≠ 0) or
    (EPCM(DS:RBX).BLOCKED ≠ 0) or (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0xFFF)) )
    THEN #PF(DS:RBX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero )
    THEN #GP(0); FI;

(* Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN #GP(0); FI;

(* Re-Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))
    THEN #PF(DS:RCX); FI;

(* Check for misconfigured SECINFO flags*)
IF ( (EPCM(DS:RCX).R = 0) and (SCRATCH_SECINFO.FLAGS.R = 0) and (SCRATCH_SECINFO.FLAGS.W ≠ 0) )
    THEN #GP(0); FI;
```

```
(* Update EPCM permissions *)
EPCM(DS:RCX).R := EPCM(DS:RCX).R | SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := EPCM(DS:RCX).W | SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := EPCM(DS:RCX).X | SCRATCH_SECINFO.FLAGS.X;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.

EREPORT—Create a Cryptographic Report of the Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLU[EREPORT]	IR	V/V	SGX1	This leaf function creates a cryptographic report of the enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX	RDX
IR	EREPORT (In)	Address of TARGETINFO (In)	Address of REPORTDATA (In)	Address where the REPORT is written to in an OUTPUTDATA (In)

Description

This leaf function creates a cryptographic REPORT that describes the contents of the enclave. This instruction leaf can only be executed when inside the enclave. The cryptographic report can be used by other enclaves to determine that the enclave is running on the same platform.

RBX contains the effective address of the MRENCLAVE value of the enclave that will authenticate the REPORT output, using the REPORT key delivered by EGETKEY command for that enclave. RCX contains the effective address of a 64-byte REPORTDATA structure, which allows the caller of the instruction to associate data with the enclave from which the instruction is called. RDX contains the address where the REPORT will be output by the instruction.

EREPORT Memory Parameter Semantics

TARGETINFO	REPORTDATA	OUTPUTDATA
Read access by Enclave	Read access by Enclave	Read/Write access by Enclave

This instruction leaf perform the following:

1. Validate the 3 operands (RBX, RCX, RDX) are inside the enclave.
2. Compute a report key for the target enclave, as indicated by the value located in RBX(TARGETINFO).
3. Assemble the enclave SECS data to complete the REPORT structure (including the data provided using the RCX (REPORTDATA) operand).
4. Computes a cryptographic hash over REPORT structure.
5. Add the computed hash to the REPORT structure.
6. Output the completed REPORT structure to the address in RDX (OUTPUTDATA).

The instruction fails if the operands are not properly aligned.

CR_REPORT_KEYID, used to provide key wearout protection, is populated with a statistically unique value on boot of the platform by a trusted entity within the SGX TCB.

The instruction faults if any of the following:

EREPORT Faulting Conditions

An effective address not properly aligned.	An memory address does not resolve in an EPC page.
If accessing an invalid EPC page.	If the EPC page is blocked.
May page fault.	

Concurrency Restrictions

Table 39-75. Base Concurrency Restrictions of EREPORT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREPORT	TARGETINFO [DS:RBX]	Concurrent		
	REPORTDATA [DS:RCX]	Concurrent		
	OUTPUTDATA [DS:RDX]	Concurrent		

Table 39-76. Additional Concurrency Restrictions of EREPORT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREPORT	TARGETINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RDX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EREPORT Operational Flow

Name	Type	Size (bits)	Description
TMP_ATTRIBUTES		32	Physical address of SECS of the enclave to which source operand belongs.
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_REPORTKEY		128	REPORTKEY generated by the instruction.
TMP_REPORT		3712	

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Address verification for TARGETINFO (RBX) *)
IF ( (DS:RBX is not 512Byte Aligned) or (DS:RBX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
    (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )
    THEN #PF(DS:RBX);
FI;

(* Verify RESERVED spaces in TARGETINFO are valid *)
IF (DS:RBX.RESERVED != 0)
    THEN #GP(0); FI;

(* Address verification for REPORTDATA (RCX) *)
IF ( (DS:RCX is not 128Byte Aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
    THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).R = 0) )
    THEN #PF(DS:RCX);
FI;

(* Address verification for OUTPUTDATA (RDX) *)
IF ( (DS:RDX is not 512Byte Aligned) or (DS:RDX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

IF (EPCM(DS:RDX).VALID = 0)
    THEN #PF(DS:RDX); FI;

IF (EPCM(DS:RDX).BLOCKED = 1)
    THEN #PF(DS:RDX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RDX).PT ≠ PT_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RDX).ENCLAVEADDRESS ≠ (DS:RDX & ~0FFFH) ) or (EPCM(DS:RDX).W = 0) )
    THEN #PF(DS:RDX);
FI;

(* REPORT MAC needs to be computed over data which cannot be modified *)
TMP_REPORT.CPUSVN := CR_CPUSVN;
TMP_REPORT.ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
TMP_REPORT.ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
TMP_REPORT.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
TMP_REPORT.ISVSVN := TMP_CURRENTSECS.ISVSVN;
TMP_REPORT.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;
TMP_REPORT.REPORTDATA := DS:RCX[511:0];
TMP_REPORT.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
TMP_REPORT.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_REPORT.MRRESERVED := 0;
TMP_REPORT.KEYID[255:0] := CR_REPORT_KEYID;
TMP_REPORT.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
TMP_REPORT.CONFIGID := TMP_CURRENTSECS.CONFIGID;
TMP_REPORT.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN TMP_REPORT.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES; FI;

(* Derive the report key *)
TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := 0;
TMP_KEYDEPENDENCIES.ISVSVN := 0;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := DS:RBX.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := DS:RBX.MEASUREMENT;
TMP_KEYDEPENDENCIES.MRSIGNER := 0;
TMP_KEYDEPENDENCIES.KEYID := TMP_REPORT.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := DS:RBX.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := DS:RBX.CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := DS:RBX.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := 0;
FI;

(* Calculate the derived key*)
TMP_REPORTKEY := derivekey(TMP_KEYDEPENDENCIES);

(* call cryptographic CMAC function, CMAC data are not including MAC&KEYID *)
TMP_REPORT.MAC := cmac(TMP_REPORTKEY, TMP_REPORT[3071:0] );
DS:RDX[3455: 0] := TMP_REPORT;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If the address in RCS is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RCX is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

ERESUME—Re-Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLU[ERESUME]	IR	V/V	SGX1	This leaf function is used to re-enter an enclave after an interrupt.

Instruction Operand Encoding

Op/En	RAX	RBX	RCX
IR	ERESUME (In)	Address of a TCS (In)	Address of AEP (In)

Description

The ENCLU[ERESUME] instruction resumes execution of an enclave that was interrupted due to an exception or interrupt, using the machine state previously stored in the SSA.

ERESUME Memory Parameter Semantics

TCS
Enclave read/write access

The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use by another enclave.	Either of TCS-specified FS and GS segment is not a subset of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
Offsets 520-535 of the XSAVE area not 0.	The bit vector stored at offset 512 of the XSAVE area must be a subset of SECS.ATTRIBUTES.XFRM.
The SSA frame is not valid or in use.	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

The following operations are performed by ERESUME:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or an asynchronous exit due to any Interrupt event.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 41.1.2):
  - — On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 41.2.5).
  - — On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 41.2.3).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.

- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 41.2.3):
  - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
  - PEBS is suppressed.
  - AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set.
  - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Table 39-77. Base Concurrency Restrictions of ERESUME

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERESUME	TCS [DS:RBX]	Shared	#GP	

Table 39-78. Additional Concurrency Restrictions of ERESUME

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERESUME	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ERESUME Operational Flow

Name	Type	Size	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_TARGET	Effective Address	32/64	Address of first instruction inside enclave at which execution is to resume.
TMP_SECS	Effective Address	32/64	Physical address of SECS for this enclave.
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.
TMP_BRANCH_RECORD	LBR Record		From/to addresses to be pushed onto the LBR stack.
TMP_NOTIFY	Boolean	1	When set to 1, deliver an AEX notification.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or DS[bits 11:9] != 001B))
```

```
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
    THEN
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
        IF(ES usable and ES.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical))
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions are operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS))
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned))
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Make sure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFFFFFFFFFCH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
```

```
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT))
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
FI;

IF ( (DS:RBX).CSSA.FLAGS.DBGOPTIN = 0) and (DS:RBX).CSSA.FLAGS.AEXNOTIFY ≠ TMP_SECS.ATTRIBUTES.AEXNOTIFY))
    THEN #GP(0); FI;

(* Make sure the SSA contains at least one active frame *)
IF ( (DS:RBX).CSSA = 0)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * ( (DS:RBX).CSSA - 1);
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE_.MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
```

```
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA := Physical_Address (DS: TMP_GPR);

IF ((DS:RBX).FLAGS.AEXNOTIFY = 1) and (DS:TMP_GPR.AEXNOTIFY[0] = 1))
    THEN
        TMP_NOTIFY := 1;
    ELSE
        TMP_NOTIFY := 0;
FI;

IF (TMP_NOTIFY = 1)
    THEN
        (* Make sure the SSA contains at least one more frame *)
        IF ((DS:RBX).CSSA ≥ (DS:RBX).NSSA)
            THEN #GP(0); FI;

        TMP_SSA := TMP_SSA + 4096 * TMP_SECS.SSAFRAMESIZE;
        TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

        FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
            (* Check page is read/write accessible *)
            Check that DS:TMP_SSA_PAGE is read/write accessible;
            If a fault occurs, release locks, abort and deliver that fault;

            IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or
            (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF ((EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or
            (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
            (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
            (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0))
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
        ENDFOR
        (* Compute address of GPR area*)
        TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
        If a fault occurs; release locks, abort and deliver that fault;

        IF (DS:TMP_GPR does not resolve to EPC page)
            THEN #PF(DS:TMP_GPR); FI;
        IF (EPCM(DS:TMP_GPR).VALID = 0)
            THEN #PF(DS:TMP_GPR); FI;
        IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
            THEN #PF(DS:TMP_GPR); FI;
        IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
            THEN #PF(DS:TMP_GPR); FI;
        IF ((EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or
        (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
        (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
            THEN #PF(DS:TMP_GPR); FI;

        IF (TMP_MODE64 = 0)
            THEN
                IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
        FI;

        CR_GPR_PA := Physical_Address (DS: TMP_GPR);

        TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
    ELSE
        TMP_TARGET := (DS:TMP_GPR).RIP;
FI;

IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
```

```
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        IF (TMP_NOTIFY = 1)
            THEN
                TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
                TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
            ELSE
                TMP_FSBASE := DS:TMP_GPR.FSBASE;
                TMP_GSBASE := DS:TMP_GPR.GSBASE;
        FI;
        IF ((TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
            THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE))
    THEN #GP(0); FI;

TMP_IA32_U_CET := 0
TMP_SSP := 0

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        IF ( CR4.CET = 0 )
            THEN
                (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
                IF (TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) #GP(0); FI;
        FI;
        (* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail ERESUME *)
        IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1)
            THEN
                IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;
        FI;

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
    THEN
        (* Setup CET state from SECS, note tracker goes to IDLE *)
        TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
        IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1)
            THEN
                TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;
                TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
        FI;

        (* Compute linear address of what will become new CET state save area and cache its PA *)
        IF (TMP_NOTIFY = 1)
            THEN
                TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16;
            ELSE
                TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA - 1) * 16;
        FI;
        TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

        Check the TMP_CET_SAVE_PAGE page is read/write accessible
        If fault occurs release locks, abort and deliver fault

        (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
        IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
            THEN
                #PF(DS:TMP_CET_SAVE_PAGE);
        FI;

        CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)
        IF (TMP_NOTIFY = 1)
            THEN
                IF TMP_IA32_U_CET.SH_STK_EN = 1
                    THEN TMP_SSP = TCS.PREVSSP; FI;
            ELSE
                TMP_SSP = CR_CET_SAVE_AREA_PA.SSP
                TMP_IA32_U_CET.TRACKER = CR_CET_SAVE_AREA_PA.TRACKER;
                TMP_IA32_U_CET.SUPPRESS = CR_CET_SAVE_AREA_PA.SUPPRESS;
                IF ( (TMP_MODE64 = 1 AND TMP_SSP is not canonical) OR
                     (TMP_MODE64 = 0 AND (TMP_SSP & 0xFFFFFFFF00000000) ≠ 0) OR
                    (TMP_SSP is not 4 byte aligned) OR
                    (TMP_IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH AND TMP_IA32_U_CET.SUPPRESS = 1) OR
                    (CR_CET_SAVE_AREA_PA.Reserved ≠ 0) ) #GP(0); FI;
                FI;
        FI;
FI;

IF (TMP_NOTIFY = 0)
    THEN
        (* SECS.ATTRIBUTES.XFRM selects the features to be saved. *)
        (* CR_XSAVE_PAGE_n: A list of 1 or more physical address of pages that contain the XSAVE area. *)
        XRSTOR(TMP_MODE64, SECS.ATTRIBUTES.XFRM, CR_XSAVE_PAGE_n);

        IF (XRSTOR failed with #GP)
            THEN
                DS:RBX.STATE := INACTIVE;
                #GP(0);
        FI;
FI;

CR_ENCLAVE_MODE := 1;
CR_ACTIVE_SECS := TMP_SECS;
CR_ELRANGE := (TMP_SECS.BASEADDR, TMP_SECS.SIZE);
```

```
(* Save sate for possible AEXs *)
CR_TCS_PA := Physical_Address (DS:RBX);
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;

IF (TMP_NOTIFY = 1)
    THEN
        (* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
        IF (CR4.OSXSAVE = 1)
            THEN
                CR_SAVE_XCR0 := XCR0;
                XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
        FI;
FI;

RIP := TMP_TARGET;

IF (TMP_NOTIFY = 1)
    THEN
        RCX := RIP;
        RAX := (DS:RBX).CSSA;
        (* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
        DS:TMP_SSA.U_RSP := RSP;
        DS:TMP_SSA.U_RBP := RBP;
    ELSE
        Restore_GPRs from DS:TMP_GPR;

        (*Restore the RFLAGS values from SSA*)
        RFLAGS.CF := DS:TMP_GPR.RFLAGS.CF;
        RFLAGS.PF := DS:TMP_GPR.RFLAGS.PF;
        RFLAGS.AF := DS:TMP_GPR.RFLAGS.AF;
        RFLAGS.ZF := DS:TMP_GPR.RFLAGS.ZF;
        RFLAGS.SF := DS:TMP_GPR.RFLAGS.SF;
        RFLAGS.DF := DS:TMP_GPR.RFLAGS.DF;
        RFLAGS.OF := DS:TMP_GPR.RFLAGS.OF;
        RFLAGS.NT := DS:TMP_GPR.RFLAGS.NT;
        RFLAGS.AC := DS:TMP_GPR.RFLAGS.AC;
        RFLAGS.ID := DS:TMP_GPR.RFLAGS.ID;
        RFLAGS.RF := DS:TMP_GPR.RFLAGS.RF;
        RFLAGS.VM := 0;
        IF (RFLAGS.IOPL = 3)
            THEN RFLAGS.IF := DS:TMP_GPR.RFLAGS.IF; FI;

        IF (TCS.FLAGS.OPTIN = 0)
```

```
                THEN RFLAGS.TF := 0; FI;

        (* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
        IF (CR4.OSXSAVE = 1)
            THEN
                CR_SAVE_XCR0 := XCR0;
                XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
        FI;

        (* Pop the SSA stack*)
        (DS:RBX).CSSA := (DS:RBX).CSSA -1;
FI;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS[bit 9];
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS[bit 21];
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS[bit 9];
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS[bit 21];
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress all code breakpoints that are outside ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress all code breakpoints that overlap with ELRANGE;
        CR_SAVE_TF := RFLAGS.TF;
        RFLAGS.TF := 0;
        Suppress any MTF VM exits during execution of the enclave;
        Clear all pending debug exceptions;
        Clear any pending MTF VM exit;
    ELSE
```

```
        IF (TMP_NOTIFY = 1)
            THEN
                IF RFLAGS.TF = 1
                    THEN pend a single-step #DB at the end of ERESUME; FI;
                IF the "monitor trap flag" VM-execution control is set
                    THEN pend an MTF VM exit at the end of ERESUME; FI;
        ELSE
            Clear all pending debug exceptions;
            Clear pending MTF VM exits;
        FI;
FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET := IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP := SSP
                SSP := TMP_SSP;
        FI;
        IA32_U_CET := TMP_IA32_U_CET;
FI;

(* Assure consistent translations *)
Flush_linear_context;
Clear_Monitor_FSM;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment.
	If any reserved field in the TCS FLAG is set.
	If the target address is not within the CS segment.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.
#PF(error code)	If a page fault occurs in accessing memory.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment.
	If any reserved field in the TCS FLAG is set.
	If the target address is not canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

39.5 INTEL® SGX VIRTUALIZATION LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLV instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EDECVIRTCHILD—Decrement VIRTCHILDCNT in SECS

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLV[EDECVIRTCHILD]	IR	V/V	EAX[5]	This leaf function decrements the SECS VIRTCHILDCNT field.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EDECVIRTCHILD (In)	Return error code (Out)	Address of an enclave page (In)	Address of an SECS page (In)

Description

This instruction decrements the SECS VIRTCHILDCNT field. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

EDECVIRTCHILD Memory Parameter Semantics

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EDECVIRTCHILD Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

Concurrency Restrictions

Table 39-79. Base Concurrency Restrictions of EDECVIRTCHILD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDECVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_ CONFLICT	
	SECS [DS:RCX]	Concurrent		

Table 39-80. Additional Concurrency Restrictions of EDECVIRTCHILD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDECVIRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDECVIRTCHILD Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_VIRTCHILDCNT	Integer	64	Number of virtual child pages.

EDECVIRTCHILD Return Value in RAX

Error	Value	Description
No Error	0	EDECVIRTCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_INVALID_COUNTER		Attempt to decrement counter that is already zero.

```
(* check alignment of DS:RBX *)
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;

(* check DS:RBX is an linear address of an EPC page *)
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check DS:RCX is an linear address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;

(* Check the EPCPAGE for concurrency *)
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check that the EPC page is valid *)
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))
     THEN
    (* get the SECS of DS:RBX *)
    TMP_SECS := Address of SECS for (DS:RBX);
ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
    (* get the physical address of DS:RBX *)
    TMP_SECS := Physical_Address(DS:RBX);
ELSE
    (* EDECVIRTCHILD called on page of incorrect type *)
    #PF(DS:RBX, PFEC.SGX); FI;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
    #GP(0); FI;

(* Atomically decrement virtchild counter and check for underflow *)
Locked_Decrement(SECS(TMP_SECS).VIRTCHILDCNT);
IF (There was an underflow) THEN
    Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_COUNTER;
    goto DONE;
FI;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;
```

Flags Affected

ZF is set if EDECVIRTCHILD fails due to concurrent operation with another SGX instruction, or if there is a VIRT-CHILDCNT underflow. Otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

EINCVIRTCHILD—Increment VIRTCHILDCNT in SECS

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLV[EINCVIRTCHILD]	IR	V/V	EAX[5]	This leaf function increments the SECS VIRTCHILDCNT field.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EINCVIRTCHILD (In)	Return error code (Out)	Address of an enclave page (In)	Address of an SECS page (In)

Description

This instruction increments the SECS VIRTCHILDCNT field. This instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create a linear address. Segment override is not supported.

EINCVIRTCHILD Memory Parameter Semantics

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EINCVIRTCHILD Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

Concurrency Restrictions

Table 39-81. Base Concurrency Restrictions of EINCVIRTCHILD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINCVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS [DS:RCX]	Concurrent		

Table 39-82. Additional Concurrency Restrictions of EINCVIRTCHILD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINCVIRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EINCVIRTCHILD Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

EINCVIRTCHILD Return Value in RAX

Error	Value	Description
No Error	0	EINCVIRTCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

```
(* check alignment of DS:RBX *)
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;

(* check DS:RBX is an linear address of an EPC page *)
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check DS:RCX is an linear address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;

(* Check the EPCPAGE for concurrency *)
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check that the EPC page is valid *)
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))
```

```
      THEN
    (* get the SECS of DS:RBX *)
    TMP_SECS := Address of SECS for (DS:RBX);
ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
    (* get the physical address of DS:RBX *)
    TMP_SECS := Physical_Address(DS:RBX);
ELSE
    (* EINCVIRTCHILD called on page of incorrect type *)
    #PF(DS:RBX, PFEC.SGX); FI;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
    #GP(0); FI;

(* Atomically increment virtchild counter *)
Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);

RFLAGS.ZF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;
```

Flags Affected

ZF is set if EINCVIRTCHILD fails due to concurrent operation with another SGX instruction; otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

ESETCONTEXT—Set the ENCLAVECONTEXT Field in SECS

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 02H ENCLV[ESETCONTEXT]	IR	V/V	EAX[5]	This leaf function sets the ENCLAVECONTEXT field in SECS.

Instruction Operand Encoding

Op/En	EAX		RCX	RDX
IR	ESETCONTEXT (In)	Return error code (Out)	Address of the destination EPC page (In, EA)	Context Value (In, EA)

Description

The ESETCONTEXT leaf overwrites the ENCLAVECONTEXT field in the SECS. ECREATE and ELD of an SECS set the ENCLAVECONTEXT field in the SECS to the address of the SECS (for access later in ERDINFO). The ESETCONTEXT instruction allows a VMM to overwrite the default context value if necessary, for example, if the VMM is emulating ECREATE or ELD on behalf of the guest.

The content of RCX is an effective address of the SECS page to be updated, RDX contains the address pointing to the value to be stored in the SECS. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if:

- The operand is not properly aligned.
- RCX does not refer to an SECS page.

ESETCONTEXT Memory Parameter Semantics

EPCPAGE	CONTEXT
Read access permitted by Enclave	Read/Write access permitted by Non Enclave

The instruction faults if any of the following:

ESETCONTEXT Faulting Conditions

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

Concurrency Restrictions

Table 39-83. Base Concurrency Restrictions of ESETCONTEXT

Leaf	**Parameter**	**Base Concurrency Restrictions**		
		Access	**On Conflict**	**SGX_CONFLICT VM Exit Qualification**
ESETCONTEXT	SECS [DS:RCX]	Shared	SGX_EPC_PAGE_ CONFLICT	

Table 39-84. Additional Concurrency Restrictions of ESETCONTEXT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ESETCONTEXT	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ESETCONTEXT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_CONTEXT	CONTEXT	64	Data Value of CONTEXT.

ESETCONTEXT Return Value in RAX

Error	Value	Description
No Error	0	ESETCONTEXT Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

```
(* check alignment of the EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); FI;

 (* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within an EPC)THEN
    #PF(DS:RCX, PFEC.SGX); FI;

(* check alignment of the CONTEXT field (RDX) *)
IF (DS:RDX is not 8Byte Aligned) THEN
    #GP(0); FI;

 (* Load CONTEXT into local variable *)
TMP_CONTEXT := DS:RDX

(* Check the EPC page for concurrency *)
IF (EPC page is being modified) THEN
    RFLAGS.ZF := 1;
    RFLAGS.CF := 0;
    RAX := SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check page validity *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    #PF(DS:RCX, PFEC.SGX);
FI;

(* check EPC page is an SECS page *)
```

```
IF (EPCM(DS:RCX).PT is not PT_SECS) THEN
    #PF(DS:RCX, PFEC.SGX);
FI;

(* load the context value into SECS(DS:RCX).ENCLAVECONTEXT *)
SECS(DS:RCX).ENCLAVECONTEXT := TMP_CONTEXT;

RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0;
```

Flags Affected

ZF is set if ESETCONTEXT fails due to concurrent operation with another SGX instruction; otherwise cleared.

CF, PF, AF, OF, and SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

16. Updates Appendix B, Volume 3D

Change bars and violet text show changes to Appendix B of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter:

- Added MSR data field to Table B-5. "Encodings for 64-Bit Read-Only Data Fields (0010_01xx_xxxx_xxxAb)" in Section B.2.2, “64-Bit Read-Only Data Fields.”

APPENDIX B FIELD ENCODING IN VMCS

Every component of the VMCS is encoded by a 32-bit field that can be used by VMREAD and VMWRITE. Section 26.11.2 describes the structure of the encoding space (the meanings of the bits in each 32-bit encoding).

This appendix enumerates all fields in the VMCS and their encodings. Fields are grouped by width (16-bit, 32-bit, etc.) and type (guest-state, host-state, etc.).

B.1 16-BIT FIELDS

A value of 0 in bits 14:13 of an encoding indicates a 16-bit field. Only guest-state areas and the host-state area contain 16-bit fields. As noted in Section 26.11.2, each 16-bit field allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.1.1 16-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-1 enumerates the 16-bit control fields.

Table B-1. Encoding for 16-Bit Control Fields (0000_00xx_xxxx_xxx0B)

Field Name	Index	Encoding
Virtual-processor identifier (VPID)[1]	000000000B	00000000H
Posted-interrupt notification vector[2]	000000001B	00000002H
EPTP index[3]	000000010B	00000004H
HLAT prefix size[4]	000000011B	00000006H
Last PID-pointer index[5]	000000100B	00000008H

NOTES:

1. This field exists only on processors that support the 1-setting of the “enable VPID” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “process posted interrupts” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.
4. This field exists only on processors that support the 1-setting of the “enable HLAT” VM-execution control.
5. This field exists only on processors that support the 1-setting of the “IPI virtualization” VM-execution control.

B.1.2 16-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-2 enumerates 16-bit guest-state fields.

Table B-2. Encodings for 16-Bit Guest-State Fields (0000_10xx_xxxx_xxx0B)

Field Name	Index	Encoding
Guest ES selector	000000000B	00000800H
Guest CS selector	000000001B	00000802H
Guest SS selector	000000010B	00000804H
Guest DS selector	000000011B	00000806H
Guest FS selector	000000100B	00000808H

Table B-2. Encodings for 16-Bit Guest-State Fields (0000_10xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Guest GS selector	000000101B	0000080AH
Guest LDTR selector	000000110B	0000080CH
Guest TR selector	000000111B	0000080EH
Guest interrupt status[1]	000001000B	00000810H
PML index[2]	000001001B	00000812H
Guest UINV[3]	000001010B	00000814H

NOTES:

1. This field exists only on processors that support the 1-setting of the "virtual-interrupt delivery" VM-execution control.
2. This field exists only on processors that support the 1-setting of the "enable PML" VM-execution control.
3. This field exists only on processors that support the 1-setting of either the "clear UINV" VM-exit control or the "load UINV" VM-entry control.

B.1.3 16-Bit Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-3 enumerates the 16-bit host-state fields.

Table B-3. Encodings for 16-Bit Host-State Fields (0000_11xx_xxxx_xxx0B)

Field Name	Index	Encoding
Host ES selector	000000000B	00000C00H
Host CS selector	000000001B	00000C02H
Host SS selector	000000010B	00000C04H
Host DS selector	000000011B	00000C06H
Host FS selector	000000100B	00000C08H
Host GS selector	000000101B	00000C0AH
Host TR selector	000000110B	00000C0CH

B.2 64-BIT FIELDS

A value of 1 in bits 14:13 of an encoding indicates a 64-bit field. There are 64-bit fields only for controls and for guest state. As noted in Section 26.11.2, every 64-bit field has two encodings, which differ on bit 0, the access type. Thus, each such field has an even encoding for full access and an odd encoding for high access.

B.2.1 64-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-4 enumerates the 64-bit control fields.

Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb)

Field Name	Index	Encoding
Address of I/O bitmap A (full)	000000000B	00002000H
Address of I/O bitmap A (high)		00002001H
Address of I/O bitmap B (full)	000000001B	00002002H
Address of I/O bitmap B (high)		00002003H

Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb) (Contd.)

Field Name	Index	Encoding
Address of MSR bitmaps (full)[1]	000000010B	00002004H
Address of MSR bitmaps (high)[1]		00002005H
VM-exit MSR-store address (full)	000000011B	00002006H
VM-exit MSR-store address (high)		00002007H
VM-exit MSR-load address (full)	000000100B	00002008H
VM-exit MSR-load address (high)		00002009H
VM-entry MSR-load address (full)	000000101B	0000200AH
VM-entry MSR-load address (high)		0000200BH
Executive-VMCS pointer (full)	000000110B	0000200CH
Executive-VMCS pointer (high)		0000200DH
PML address (full)[2]	000000111B	0000200EH
PML address (high)[2]		0000200FH
TSC offset (full)	000001000B	00002010H
TSC offset (high)		00002011H
Virtual-APIC address (full)[3]	000001001B	00002012H
Virtual-APIC address (high)[3]		00002013H
APIC-access address (full)[4]	000001010B	00002014H
APIC-access address (high)[4]		00002015H
Posted-interrupt descriptor address (full)[5]	000001011B	00002016H
Posted-interrupt descriptor address (high)[5]		00002017H
VM-function controls (full)[6]	000001100B	00002018H
VM-function controls (high)[6]		00002019H
EPT pointer (EPTP; full)[7]	000001101B	0000201AH
EPT pointer (EPTP; high)[7]		0000201BH
EOI-exit bitmap 0 (EOI_EXIT0; full)[8]	000001110B	0000201CH
EOI-exit bitmap 0 (EOI_EXIT0; high)[8]		0000201DH
EOI-exit bitmap 1 (EOI_EXIT1; full)[8]	000001111B	0000201EH
EOI-exit bitmap 1 (EOI_EXIT1; high)[8]		0000201FH
EOI-exit bitmap 2 (EOI_EXIT2; full)[8]	000010000B	00002020H
EOI-exit bitmap 2 (EOI_EXIT2; high)[8]		00002021H
EOI-exit bitmap 3 (EOI_EXIT3; full)[8]	000010001B	00002022H
EOI-exit bitmap 3 (EOI_EXIT3; high)[8]		00002023H
EPTP-list address (full)[9]	000010010B	00002024H
EPTP-list address (high)[9]		00002025H
VMREAD-bitmap address (full)[10]	000010011B	00002026H
VMREAD-bitmap address (high)[10]		00002027H
VMWRITE-bitmap address (full)[10]	000010100B	00002028H
VMWRITE-bitmap address (high)[10]		00002029H

Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb) (Contd.)

Field Name	Index	Encoding
Virtualization-exception information address (full)[11]	000010101B	0000202AH
Virtualization-exception information address (high)[11]		0000202BH
XSS-exiting bitmap (full)[12]	000010110B	0000202CH
XSS-exiting bitmap (high)[12]		0000202DH
ENCLS-exiting bitmap (full)[13]	000010111B	0000202EH
ENCLS-exiting bitmap (high)[13]		0000202FH
Sub-page-permission-table pointer (full)[14]	000011000B	00002030H
Sub-page-permission-table pointer (high)[14]		00002031H
TSC multiplier (full)[15]	000011001B	00002032H
TSC multiplier (high)[15]		00002033H
Tertiary processor-based VM-execution controls (full)[16]	000011010B	00002034H
Tertiary processor-based VM-execution controls (high)[16]		00002035H
ENCLV-exiting bitmap (full)[17]	000011011B	00002036H
ENCLV-exiting bitmap (high)[17]		00002037H
Low PASID directory address (full)[18]	000011100B	00002038H
Low PASID directory address (high)[18]		00002039H
High PASID directory address (full)[18]	000011101B	0000203AH
High PASID directory address (high)[18]		0000203BH
Shared EPT pointer (full)[19]	000011110B	0000203CH
Shared EPT pointer (high)[19]		0000203DH
PCONFIG-exiting bitmap (full)[20]	000011111B	0000203EH
PCONFIG-exiting bitmap (high)[20]		0000203FH
Hypervisor-managed linear-address translation pointer (HLATP; full)[21]	000100000B	00002040H
HLATP (high)[21]		00002041H
PID-pointer table address (full)[22]	000100001B	00002042H
PID-pointer table address (high)[22]		00002043H
Secondary VM-exit controls (full)[23]	000100010B	00002044H
Secondary VM-exit controls (high)[23]		00002045H
IA32_SPEC_CTRL mask (full)[24]	000100101B	0000204AH
IA32_SPEC_CTRL mask (high)[24]		0000204BH
IA32_SPEC_CTRL shadow (full)[24]	000100110B	0000204CH
IA32_SPEC_CTRL shadow (high)[24]		0000204DH

NOTES:

1. This field exists only on processors that support the 1-setting of the “use MSR bitmaps” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “enable PML” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.
4. This field exists only on processors that support the 1-setting of the “virtualize APIC accesses” VM-execution control.
5. This field exists only on processors that support the 1-setting of the “process posted interrupts” VM-execution control.
6. This field exists only on processors that support the 1-setting of the “enable VM functions” VM-execution control.
7. This field exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.

8. This field exists only on processors that support the 1-setting of the “virtual-interrupt delivery” VM-execution control.
9. This field exists only on processors that support the 1-setting of the “EPTP switching” VM-function control.
10. This field exists only on processors that support the 1-setting of the “VMCS shadowing” VM-execution control.
11. This field exists only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.
12. This field exists only on processors that support the 1-setting of the “enable XSAVES/XRSTORS” VM-execution control.
13. This field exists only on processors that support the 1-setting of the “enable ENCLS exiting” VM-execution control.
14. This field exists only on processors that support the 1-setting of the “sub-page write permissions for EPT” VM-execution control.
15. This field exists only on processors that support the 1-setting of the “use TSC scaling” VM-execution control.
16. This field exists only on processors that support the 1-setting of the “activate tertiary controls” VM-execution control.
17. This field exists only on processors that support the 1-setting of the “enable ENCLV exiting” VM-execution control.
18. This field exists only on processors that support the 1-setting of the “PASID translation” VM-execution control.
19. This field exists only on processors that support the 1-setting of the “shared-EPTP” VM-execution control.
20. This field exists only on processors that support the 1-setting of the “enable PCONFIG” VM-execution control.
21. This field exists only on processors that support the 1-setting of the “enable HLAT” VM-execution control.
22. This field exists only on processors that support the 1-setting of the “IPI virtualization” VM-execution control.
23. This field exists only on processors that support the 1-setting of the “activate secondary controls” VM-exit control.
24. This field exists only on processors that support the 1-setting of the “virtualize IA32_SPEC_CTRL” VM-execution control.

B.2.2 64-Bit Read-Only Data Fields

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. Table B-5 enumerates the 64-bit read-only data fields.

Table B-5. Encodings for 64-Bit Read-Only Data Fields (0010_01xx_xxxx_xxxAb)

Field Name	Index	Encoding
Guest-physical address (full)[1]	000000000B	00002400H
Guest-physical address (high)[1]		00002401H
MSR data (full)[2]	000000001B	00002402H
MSR data (high)[2]		00002403H

NOTES:
1. This field exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “enable MSR-list instructions” VM-execution control.

B.2.3 64-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-6 enumerates the 64-bit guest-state fields.

Table B-6. Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx_xxxAb)

Field Name	Index	Encoding
VMCS link pointer (full)	000000000B	00002800H
VMCS link pointer (high)		00002801H
Guest IA32_DEBUGCTL (full)	000000001B	00002802H
Guest IA32_DEBUGCTL (high)		00002803H
Guest IA32_PAT (full)[1]	000000010B	00002804H
Guest IA32_PAT (high)[1]		00002805H

Table B-6. Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx_xxxAb) (Contd.)

Field Name	Index	Encoding
Guest IA32_EFER (full)[2]	000000011B	00002806H
Guest IA32_EFER (high)[2]		00002807H
Guest IA32_PERF_GLOBAL_CTRL (full)[3]	000000100B	00002808H
Guest IA32_PERF_GLOBAL_CTRL (high)[3]		00002809H
Guest PDPTE0 (full)[4]	000000101B	0000280AH
Guest PDPTE0 (high)[4]		0000280BH
Guest PDPTE1 (full)[4]	000000110B	0000280CH
Guest PDPTE1 (high)[4]		0000280DH
Guest PDPTE2 (full)[4]	000000111B	0000280EH
Guest PDPTE2 (high)[4]		0000280FH
Guest PDPTE3 (full)[4]	000001000B	00002810H
Guest PDPTE3 (high)[4]		00002811H
Guest IA32_BNDCFGS (full)[5]	000001001B	00002812H
Guest IA32_BNDCFGS (high)[5]		00002813H
Guest IA32_RTIT_CTL (full)[6]	000001010B	00002814H
Guest IA32_RTIT_CTL (high)[6]		00002815H
Guest IA32_LBR_CTL (full)[7]	000001011B	00002816H
Guest IA32_LBR_CTL (high)[7]		00002817H
Guest IA32_PKRS (full)[8]	000001100B	00002818H
Guest IA32_PKRS (high)[8]		00002819H

NOTES:

1. This field exists only on processors that support either the 1-setting of the “load IA32_PAT” VM-entry control or that of the “save IA32_PAT” VM-exit control.
2. This field exists only on processors that support either the 1-setting of the “load IA32_EFER” VM-entry control or that of the “save IA32_EFER” VM-exit control.
3. This field exists only on processors that support the 1-setting of the “load IA32_PERF_GLOBAL_CTRL” VM-entry control.
4. This field exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.
5. This field exists only on processors that support either the 1-setting of the “load IA32_BNDCFGS” VM-entry control or that of the “clear IA32_BNDCFGS” VM-exit control.
6. This field exists only on processors that support either the 1-setting of the “load IA32_RTIT_CTL” VM-entry control or that of the “clear IA32_RTIT_CTL” VM-exit control.
7. This field exists only on processors that support either the 1-setting of the “load IA32_LBR_CTL” VM-entry control or that of the “clear IA32_LBR_CTL” VM-exit control.
8. This field exists only on processors that support the 1-setting of the “load PKRS” VM-entry control.

B.2.4 64-Bit Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-7 enumerates the 64-bit control fields.

Table B-7. Encodings for 64-Bit Host-State Fields (0010_11xx_xxxx_xxxAb)

Field Name	Index	Encoding
Host IA32_PAT (full)[1]	000000000B	00002C00H
Host IA32_PAT (high)[1]		00002C01H
Host IA32_EFER (full)[2]	000000001B	00002C02H
Host IA32_EFER (high)[2]		00002C03H
Host IA32_PERF_GLOBAL_CTRL (full)[3]	000000010B	00002C04H
Host IA32_PERF_GLOBAL_CTRL (high)[3]		00002C05H
Host IA32_PKRS (full)[4]	000000011B	00002C06H
Host IA32_PKRS (high)[4]		00002C07H

NOTES:

1. This field exists only on processors that support the 1-setting of the “load IA32_PAT” VM-exit control.
2. This field exists only on processors that support the 1-setting of the “load IA32_EFER” VM-exit control.
3. This field exists only on processors that support the 1-setting of the “load IA32_PERF_GLOBAL_CTRL” VM-exit control.
4. This field exists only on processors that support the 1-setting of the “load PKRS” VM-exit control.

B.3 32-BIT FIELDS

A value of 2 in bits 14:13 of an encoding indicates a 32-bit field. As noted in Section 26.11.2, each 32-bit field allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.3.1 32-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-8 enumerates the 32-bit control fields.

Table B-8. Encodings for 32-Bit Control Fields (0100_00xx_xxxx_xxx0B)

Field Name	Index	Encoding
Pin-based VM-execution controls	000000000B	00004000H
Primary processor-based VM-execution controls	000000001B	00004002H
Exception bitmap	000000010B	00004004H
Page-fault error-code mask	000000011B	00004006H
Page-fault error-code match	000000100B	00004008H
CR3-target count	000000101B	0000400AH
Primary VM-exit controls	000000110B	0000400CH
VM-exit MSR-store count	000000111B	0000400EH
VM-exit MSR-load count	000001000B	00004010H
VM-entry controls	000001001B	00004012H
VM-entry MSR-load count	000001010B	00004014H
VM-entry interruption-information field	000001011B	00004016H

Table B-8. Encodings for 32-Bit Control Fields (0100_00xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
VM-entry exception error code	000001100B	00004018H
VM-entry instruction length	000001101B	0000401AH
TPR threshold[1]	000001110B	0000401CH
Secondary processor-based VM-execution controls[2]	000001111B	0000401EH
PLE_Gap[3]	000010000B	00004020H
PLE_Window[3]	000010001B	00004022H
Instruction-timeout control[4]	000010010B	00004024H

NOTES:

1. This field exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “activate secondary controls” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “PAUSE-loop exiting” VM-execution control.
4. This field exists only on processors that support the 1-setting of the “instruction timeout” VM-execution control.

B.3.2 32-Bit Read-Only Data Fields

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. Table B-9 enumerates the 32-bit read-only data fields.

Table B-9. Encodings for 32-Bit Read-Only Data Fields (0100_01xx_xxxx_xxx0B)

Field Name	Index	Encoding
VM-instruction error	000000000B	00004400H
Exit reason	000000001B	00004402H
VM-exit interruption information	000000010B	00004404H
VM-exit interruption error code	000000011B	00004406H
IDT-vectoring information field	000000100B	00004408H
IDT-vectoring error code	000000101B	0000440AH
VM-exit instruction length	000000110B	0000440CH
VM-exit instruction information	000000111B	0000440EH

B.3.3 32-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-10 enumerates the 32-bit guest-state fields.

Table B-10. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B)

Field Name	Index	Encoding
Guest ES limit	000000000B	00004800H
Guest CS limit	000000001B	00004802H
Guest SS limit	000000010B	00004804H
Guest DS limit	000000011B	00004806H
Guest FS limit	000000100B	00004808H
Guest GS limit	000000101B	0000480AH

Table B-10. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Guest LDTR limit	000000110B	0000480CH
Guest TR limit	000000111B	0000480EH
Guest GDTR limit	000001000B	00004810H
Guest IDTR limit	000001001B	00004812H
Guest ES access rights	000001010B	00004814H
Guest CS access rights	000001011B	00004816H
Guest SS access rights	000001100B	00004818H
Guest DS access rights	000001101B	0000481AH
Guest FS access rights	000001110B	0000481CH
Guest GS access rights	000001111B	0000481EH
Guest LDTR access rights	000010000B	00004820H
Guest TR access rights	000010001B	00004822H
Guest interruptibility state	000010010B	00004824H
Guest activity state	000010011B	00004826H
Guest SMBASE	000010100B	00004828H
Guest IA32_SYSENTER_CS	000010101B	0000482AH
VMX-preemption timer value[1]	000010111B	0000482EH

NOTES:

1. This field exists only on processors that support the 1-setting of the “activate VMX-preemption timer” VM-execution control.

The limit fields for GDTR and IDTR are defined to be 32 bits in width even though these fields are only 16-bits wide in the Intel 64 and IA-32 architectures. VM entry ensures that the high 16 bits of both these fields are cleared to 0.

B.3.4 32-Bit Host-State Field

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. There is only one such 32-bit field as given in Table B-11.

Table B-11. Encoding for 32-Bit Host-State Field (0100_11xx_xxxx_xxx0B)

Field Name	Index	Encoding
Host IA32_SYSENTER_CS	000000000B	00004C00H

B.4 NATURAL-WIDTH FIELDS

A value of 3 in bits 14:13 of an encoding indicates a natural-width field. As noted in Section 26.11.2, each of these fields allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.4.1 Natural-Width Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-12 enumerates the natural-width control fields.

Table B-12. Encodings for Natural-Width Control Fields (0110_00xx_xxxx_xxx0B)

Field Name	Index	Encoding
CR0 guest/host mask	000000000B	00006000H
CR4 guest/host mask	000000001B	00006002H
CR0 read shadow	000000010B	00006004H
CR4 read shadow	000000011B	00006006H
CR3-target value 0	000000100B	00006008H
CR3-target value 1	000000101B	0000600AH
CR3-target value 2	000000110B	0000600CH
CR3-target value 3[1]	000000111B	0000600EH

NOTES:

1. If a future implementation supports more than 4 CR3-target values, they will be encoded consecutively following the 4 encodings given here.

B.4.2 Natural-Width Read-Only Data Fields

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. Table B-13 enumerates the natural-width read-only data fields.

Table B-13. Encodings for Natural-Width Read-Only Data Fields (0110_01xx_xxxx_xxx0B)

Field Name	Index	Encoding
Exit qualification	000000000B	00006400H
I/O RCX	000000001B	00006402H
I/O RSI	000000010B	00006404H
I/O RDI	000000011B	00006406H
I/O RIP	000000100B	00006408H
Guest-linear address	000000101B	0000640AH

B.4.3 Natural-Width Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-14 enumerates the natural-width guest-state fields.

Table B-14. Encodings for Natural-Width Guest-State Fields (0110_10xx_xxxx_xxx0B)

Field Name	Index	Encoding
Guest CR0	000000000B	00006800H
Guest CR3	000000001B	00006802H
Guest CR4	000000010B	00006804H
Guest ES base	000000011B	00006806H
Guest CS base	000000100B	00006808H
Guest SS base	000000101B	0000680AH
Guest DS base	000000110B	0000680CH
Guest FS base	000000111B	0000680EH

Table B-14. Encodings for Natural-Width Guest-State Fields (0110_10xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Guest GS base	000001000B	00006810H
Guest LDTR base	000001001B	00006812H
Guest TR base	000001010B	00006814H
Guest GDTR base	000001011B	00006816H
Guest IDTR base	000001100B	00006818H
Guest DR7	000001101B	0000681AH
Guest RSP	000001110B	0000681CH
Guest RIP	000001111B	0000681EH
Guest RFLAGS	000010000B	00006820H
Guest pending debug exceptions	000010001B	00006822H
Guest IA32_SYSENTER_ESP	000010010B	00006824H
Guest IA32_SYSENTER_EIP	000010011B	00006826H
Guest IA32_S_CET[1]	000010100B	00006828H
Guest SSP[1]	000010101B	0000682AH
Guest IA32_INTERRUPT_SSP_TABLE_ADDR[1]	000010110B	0000682CH

NOTES:

1. This field is supported only on processors that support the 1-setting of the "load CET state" VM-entry control.

The base-address fields for ES, CS, SS, and DS in the guest-state area are defined to be natural-width (with 64 bits on processors supporting Intel 64 architecture) even though these fields are only 32-bits wide in the Intel 64 architecture. VM entry ensures that the high 32 bits of these fields are cleared to 0.

B.4.4 Natural-Width Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-15 enumerates the natural-width host-state fields.

Table B-15. Encodings for Natural-Width Host-State Fields (0110_11xx_xxxx_xxx0B)

Field Name	Index	Encoding
Host CR0	000000000B	00006C00H
Host CR3	000000001B	00006C02H
Host CR4	000000010B	00006C04H
Host FS base	000000011B	00006C06H
Host GS base	000000100B	00006C08H
Host TR base	000000101B	00006C0AH
Host GDTR base	000000110B	00006C0CH
Host IDTR base	000000111B	00006C0EH
Host IA32_SYSENTER_ESP	000001000B	00006C10H
Host IA32_SYSENTER_EIP	000001001B	00006C12H
Host RSP	000001010B	00006C14H
Host RIP	000001011B	00006C16H

Table B-15. Encodings for Natural-Width Host-State Fields (0110_11xx_xxxx_xxx0B) (Contd.)

Field Name	Index	Encoding
Host IA32_S_CET[1]	000001100B	00006C18H
Host SSP[1]	000001101B	00006C1AH
Host IA32_INTERRUPT_SSP_TABLE_ADDR[1]	000001110B	00006C1CH

NOTES:

1. This field is supported only on processors that support the 1-setting of the “load CET state” VM-exit control.

17. Updates to Chapter 2, Volume 4

Change bars and violet text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4:* Model-Specific Registers.

Changes to this chapter:

- Corrected typo for Reserved field in IA32_OVERCLOCKING_STATUS MSR in Table 2-2, "IA-32 Architectural MSRs."

CHAPTER 2
MODEL-SPECIFIC REGISTERS (MSRS)

This chapter lists MSRs across Intel processor families. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions. The scope of an MSR defines the set of processors that access the same MSR with RDMSR and WRMSR. Thread-scope MSRs are unique to every logical processor. Core-scope MSRs are shared by the threads in the same core; similarly for module-scope, die-scope, and package-scope.

When a processor package contains a single die, die-scope and package-scope are synonymous. When a package contains multiple die, they are distinct.

NOTE

For information on hierarchical level types supported, refer to the CPUID Leaf 1FH definition for the actual level type numbers: "V2 Extended Topology Enumeration Leaf" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. Also see Section 10.9.1, "Hierarchical Mapping of Shared Resources," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A). Table 2-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

Table 2-1. CPUID Signature Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_BDH	Intel® Series 2 Core™ Ultra processors supporting Lunar Lake performance hybrid architecture
06_ADH, 06_AEH	Intel® Xeon® 6 P-core processors based on Granite Rapids microarchitecture
06_AFH	Intel® Xeon® 6 E-core processors based on Sierra Forest microarchitecture
06_AAH	Intel® Core™ Ultra 7 processors supporting Meteor Lake performance hybrid architecture
06_CFH	5th generation Intel® Xeon® Scalable Processor Family based on Emerald Rapids microarchitecture
06_8FH	4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture
06_BAH, 06_B7H, 06_BFH	13th generation Intel® Core™ processors supporting Raptor Lake performance hybrid architecture
06_97H, 06_9AH	12th generation Intel® Core™ processors supporting Alder Lake performance hybrid architecture
06_8CH, 06_8DH	11th generation Intel® Core™ processors based on Tiger Lake microarchitecture
06_A7H	11th generation Intel® Core™ processors based on Rocket Lake microarchitecture
06_7DH, 06_7EH	10th generation Intel® Core™ processors based on Ice Lake microarchitecture
06_A5H, 06_A6H	10th generation Intel® Core™ processors based on Comet Lake microarchitecture
06_66H	Intel® Core™ processors based on Cannon Lake microarchitecture
06_8EH, 06_9EH	7th generation Intel® Core™ processors based on Kaby Lake microarchitecture, 8th and 9th generation Intel® Core™ processors based on Coffee Lake microarchitecture, Intel® Xeon® E processors based on Coffee Lake microarchitecture
06_6AH, 06_6CH	3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture

Table 2-1. CPUID Signature Values of DisplayFamily_DisplayModel (Contd.)

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_55H	Intel® Xeon® Scalable Processor Family based on Skylake microarchitecture, 2nd generation Intel® Xeon® Scalable Processor Family based on Cascade Lake product, and 3rd generation Intel® Xeon® Scalable Processor Family based on Cooper Lake product
06_4EH, 06_5EH	6th generation Intel Core processors and Intel Xeon processor E3-1500m v5 product family and E3-1200 v5 product family based on Skylake microarchitecture
06_85H	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture
06_57H	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series based on Knights Landing microarchitecture
06_56H	Intel Xeon processor D-1500 product family based on Broadwell microarchitecture
06_4FH	Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition
06_47H	5th generation Intel Core processors, Intel Xeon processor E3-1200 v4 product family based on Broadwell microarchitecture
06_3DH	Intel Core M-5xxx Processor, 5th generation Intel Core processors based on Broadwell microarchitecture
06_3FH	Intel Xeon processor E5-4600/2600/1600 v3 product families, Intel Xeon processor E7 v3 product families based on Haswell-E microarchitecture, Intel Core i7-59xx Processor Extreme Edition
06_3CH, 06_45H, 06_46H	4th Generation Intel Core processor and Intel Xeon processor E3-1200 v3 product family based on Haswell microarchitecture
06_3EH	Intel Xeon processor E7-8800/4800/2800 v2 product families based on Ivy Bridge-E microarchitecture
06_3EH	Intel Xeon processor E5-2600/1600 v2 product families and Intel Xeon processor E5-2400 v2 product family based on Ivy Bridge-E microarchitecture, Intel Core i7-49xx Processor Extreme Edition
06_3AH	3rd Generation Intel Core Processor and Intel Xeon processor E3-1200 v2 product family based on Ivy Bridge microarchitecture
06_2DH	Intel Xeon processor E5 Family based on Sandy Bridge microarchitecture, Intel Core i7-39xx Processor Extreme Edition
06_2FH	Intel Xeon Processor E7 Family
06_2AH	Intel Xeon processor E3-1200 product family; 2nd Generation Intel Core i7, i5, i3 Processors 2xxx Series
06_2EH	Intel Xeon processor 7500, 6500 series
06_25H, 06_2CH	Intel Xeon processors 3600, 5600 series, Intel Core i7, i5, and i3 Processors
06_1EH, 06_1FH	Intel Core i7 and i5 Processors
06_1AH	Intel Core i7 Processor, Intel Xeon processor 3400, 3500, 5500 series
06_1DH	Intel Xeon processor MP 7400 series
06_17H	Intel Xeon processor 3100, 3300, 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series
06_0FH	Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_86H, 06_96H, 06_9CH	Intel Atom® processors, Intel® Celeron® processors, Intel® Pentium® processors, and Intel® Pentium® Silver processors based on Tremont Microarchitecture
06_7AH	Intel Atom processors based on Goldmont Plus microarchitecture
06_5FH	Intel Atom processors based on Goldmont microarchitecture (Denverton)
06_5CH	Intel Atom processors based on Goldmont microarchitecture

Table 2-1. CPUID Signature Values of DisplayFamily_DisplayModel (Contd.)

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_4CH	Intel Atom processor X7-Z8000 and X5-Z8000 series based on Airmont microarchitecture
06_5DH	Intel Atom processor X3-C3000 based on Silvermont microarchitecture
06_5AH	Intel Atom processor Z3500 series
06_4AH	Intel Atom processor Z3400 series
06_37H	Intel Atom processor E3000 series, Z3600 series, Z3700 series
06_4DH	Intel Atom processor C2000 series
06_36H	Intel Atom processor S1000 Series
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	Intel Atom processor family, Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series
0F_06H	Intel Xeon processor 7100, 5000 Series, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
0F_03H, 0F_04H	Intel Xeon processor, Intel Xeon processor MP, Intel Pentium 4, Pentium D processors
06_09H	Intel Pentium M processor
0F_02H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon processor, Intel Pentium III processor
06_03H, 06_05H	Intel Pentium II Xeon processor, Intel Pentium II processor
06_01H	Intel Pentium Pro processor
05_01H, 05_02H, 05_04H	Intel Pentium processor, Intel Pentium processor with MMX Technology

The Intel® Quark™ SoC X1000 processor can be identified by the signature of DisplayFamily_DisplayModel = 05_09H and SteppingID = 0

2.1 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these "architectural MSRs" were given the prefix "IA32_". Table 2-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table 2-2 and certain bit fields in an MSR address that may overlap with architectural MSR addresses are model-specific. Code that accesses a model-specific MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table 2-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of "DF_DM" (see Table 2-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as "MAXPHYADDR" in Table 2-2. "MAXPHYADDR" is reported by CPUID.8000_0008H leaf.

MSR address range between 40000000H - 4000FFFFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

Table 2-2. IA-32 Architectural MSRs

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 0H, 0	IA32_P5_MC_ADDR (P5_MC_ADDR)	
See Section 2.23, "MSRs in Pentium Processors."		Pentium Processor (05_01H)
Register Address: 1H, 1	IA32_P5_MC_TYPE (P5_MC_TYPE)	
See Section 2.23, "MSRs in Pentium Processors."		DF_DM = 05_01H
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, "Monitor/Mwait Address Range Determination."		0F_03H
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER (TSC)	
See Section 19.17, "Time-Stamp Counter."		05_01H
Register Address: 17H, 23	IA32_PLATFORM_ID (MSR_PLATFORM_ID)	
Platform ID (R/O) The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.		06_01H
49:0	Reserved.	
52:50	Platform ID (R/O) Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7	
63:53	Reserved.	
Register Address: 1BH, 27	IA32_APIC_BASE (APIC_BASE)	
This register holds the APIC base address, permitting the relocation of the APIC memory map. See Section 12.4.4, "Local APIC Status and Location," and Section 12.4.5, "Relocating the Local APIC Registers."		06_01H
7:0	Reserved.	
8	BSP Flag (R/W)	
9	Reserved.	
10	Enable x2APIC mode.	06_1AH
11	APIC Global Enable (R/W)	
(MAXPHYADDR -1):12	APIC Base (R/W)	
63: MAXPHYADDR	Reserved.	
Register Address: 2FH, 47	IA32_BARRIER	
IA32_BARRIER (R/O) The IA32_BARRIER MSR ensures ordered execution by acting like LFENCE, controlling the sequencing of subsequent MSR reads after prior MSR reads and instructions.		CPUID.07H.01H:EAX[27]=1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
31:0	DATA Reserved. Always 0.	
63:32	Reserved.	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64 Processor (R/W)		If any one enumeration condition for defined bit field holds.
0	Lock bit (R/WO): (1 = locked). When set, locks this MSR from being written; writes to this bit will result in GP(0). Note: Once the Lock bit is set, the contents of this register cannot be modified. Therefore the lock bit must be set after configuring support for Intel Virtualization Technology and prior to transferring control to an option ROM or the OS. Hence, once the Lock bit is set, the entire IA32_FEATURE_CONTROL contents are preserved across RESET when PWRGOOD is not deasserted.	If any one enumeration condition for defined bit field position greater than bit 0 holds.
1	Enable VMX inside SMX operation (R/WL) This bit enables a system executive to use VMX in conjunction with SMX to support Intel® Trusted Execution Technology. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag and SMX feature flag set (ECX bits 5 and 6 respectively).	If CPUID.01H:ECX[5] = 1 && CPUID.01H:ECX[6] = 1
2	Enable VMX outside SMX operation (R/WL) This bit enables VMX for a system executive that does not require SMX. BIOS must set this bit only when the CPUID function 1 returns the VMX feature flag set (ECX bit 5).	If CPUID.01H:ECX[5] = 1
7:3	Reserved.	
14:8	SENTER Local Function Enables (R/WL) When set, each bit in the field represents an enable control for a corresponding SENTER function. This field is supported only if CPUID.1:ECX.[bit 6] is set.	If CPUID.01H:ECX[6] = 1
15	SENTER Global Enable (R/WL) This bit must be set to enable SENTER leaf functions. This bit is supported only if CPUID.1:ECX.[bit 6] is set.	If CPUID.01H:ECX[6] = 1
16	Reserved.	
17	SGX Launch Control Enable (R/WL) This bit must be set to enable runtime re-configuration of SGX Launch Control via the IA32_SGXLEPUBKEYHASHn MSR.	If CPUID.(EAX=07H, ECX=0H): ECX[30] = 1
18	SGX Global Enable (R/WL) This bit must be set to enable SGX leaf functions.	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
19	Reserved.	
20	LMCE On (R/WL) When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.	If IA32_MCG_CAP[27] = 1
63:21	Reserved.	
Register Address: 3BH, 59	IA32_TSC_ADJUST	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Per Logical Processor TSC Adjust (R/Write to clear)		If CPUID.(EAX=07H, ECX=0H): EBX[1] = 1
63:0	THREAD_ADJUST Local offset value of the IA32_TSC for a logical processor. Reset value is zero. A write to IA32_TSC will modify the local offset in IA32_TSC_ADJUST and the content of IA32_TSC, but does not affect the internal invariant TSC hardware.	
Register Address: 48H, 72	IA32_SPEC_CTRL	
Speculation Control (R/W) The MSR bits are defined as logical processor scope. On some core implementations, the bits may impact sibling logical processors on the same core. This MSR has a value of 0 after reset and is unaffected by INIT# or SIPI#.		If any one of the enumeration conditions for defined bit field positions holds.
0	Indirect Branch Restricted Speculation (IBRS). Restricts speculation of indirect branch.	If CPUID.(EAX=07H, ECX=0):EDX[26]=1
1	Single Thread Indirect Branch Predictors (STIBP). Prevents indirect branch predictions on all logical processors on the core from being controlled by any sibling logical processor in the same core.	If CPUID.(EAX=07H, ECX=0):EDX[27]=1
2	Speculative Store Bypass Disable (SSBD) delays speculative execution of a load until the addresses for all older stores are known.	If CPUID.(EAX=07H, ECX=0):EDX[31]=1
3	IPRED_DIS_U If 1, enables IPRED_DIS control for CPL3.	If CPUID.(EAX=07H, ECX=2):EDX[1]=1
4	IPRED_DIS_S If 1, enables IPRED_DIS control for CPL0/1/2.	If CPUID.(EAX=07H, ECX=2):EDX[1]=1
5	RRSBA_DIS_U If 1, disables RRSBA behavior for CPL3.	If CPUID.(EAX=07H, ECX=2):EDX[2]=1
6	RRSBA_DIS_S If 1, disables RRSBA behavior for CPL0/1/2.	If CPUID.(EAX=07H, ECX=2):EDX[2]=1
7	PSFD If 1, disables Fast Store Forwarding Predictor. Note that setting bit 2 (SSBD) also disables this.	If CPUID.(EAX=07H, ECX=2):EDX[0]=1
8	DDPD_U If 1, disables the Data Dependent Prefetcher that examines data values in memory while CPL = 3. Note that setting bit 2 (SSBD) also disables this.	If CPUID.(EAX=07H, ECX=2):EDX[3]=1
9	Reserved.	
10	BHI_DIS_S When ‘1, enables BHI_DIS_S behavior.	If CPUID.(EAX=07H, ECX=2):EDX[4]=1
63:11	Reserved.	
Register Address: 49H, 73	IA32_PRED_CMD	
Prediction Command (WO) Gives software a way to issue commands that affect the state of predictors.		If any one of the enumeration conditions for defined bit field positions holds.

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
0	Indirect Branch Prediction Barrier (IBPB)	If CPUID.(EAX=07H, ECX=0):EDX[26]=1
63:1	Reserved.	
Register Address: 4EH, 78	IA32_PPIN_CTL	
Protected Processor Inventory Number Enable Control (R/W)		If CPUID.(EAX=07H, ECX=01H):EBX[0]=1[1]
0	LockOut (R/WO) If 0, indicates that further writes to IA32_PPIN_CTL is allowed. If 1, indicates that further writes to IA32_PPIN_CTL is disallowed. Writing 1 to this bit is only permitted if the Enable_PPIN bit is clear. The Privileged System Software Inventory Agent should read IA32_PPIN_CTL[bit 1] to determine if IA32_PPIN is accessible. The Privileged System Software Inventory Agent is not expected to write to this MSR.	
1	Enable_PPIN (R/W) If 1, indicates that IA32_PPIN is accessible using RDMSR. If 0, indicates that IA32_PPIN is inaccessible using RDMSR. Any attempt to read IA32_PPIN will cause #GP.	
63:2	Reserved.	
Register Address: 4FH, 79	IA32_PPIN	
Protected Processor Inventory Number (R/O)		If CPUID.(EAX=07H, ECX=01H):EBX[0]=1[1]
63:0	Protected Processor Inventory Number (R/O) A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to IA32_PPIN is enabled. Access to IA32_PPIN is permitted only if IA32_PPIN_CTL[bits 1:0] = ‘10b’.	
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG (BIOS_UPDT_TRIG)	
BIOS Update Trigger (W) Executing a WRMSR instruction to this MSR causes a microcode update to be loaded into the processor. See Section 11.11.6, “Microcode Update Loader.” A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.		06_01H
Register Address: 7AH, 122	IA32_FEATURE_ACTIVATION	
Feature Activation (R/W) Implements Feature Activation command. WRMSR to this address activates all ’activatable’ features on this thread.		
0	SE Secure Enclaves feature activation.	
1	KL Keylocker feature activation.	
63:2	Reserved.	
Register Address: 7BH, 123	IA32_MCU_ENUMERATION	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
IA32_MCU_ENUMERATION (R/O) Enumeration of architectural features.		
0	UNIFORM_MCU_AVAIL When set to 1, uniform microcode update is available, and UNIFORM_MCU_SCOPE (bits [10:8]) indicates the scope of writes to IA32_BIOS_UPDT_TRIG. When set to 0, uniform microcode update is not available, and writes to IA32_BIOS_UPDT_TRIG are core scoped.	
1	UNIFORM_MCU_CONFIG_REQD When set to 1, indicates that configuration is required to ensure that all MCU components are updated on WRMSR 79H, and UNIFORM_MCU_CONFIG_COMPLETE (bit 2) should be checked to determine whether the necessary configuration has been completed. When set to 0, indicates that no configuration is required, and UNIFORM_MCU_CONFIG_COMPLETE should be ignored.	
2	UNIFORM_MCU_CONFIG_COMPLETE If UNIFORM_MCU_CONFIG_REQD (bit 1) is 0, then this bit should be ignored. If UNIFORM_MCU_CONFIG_REQD is 1, then this bit indicates whether all necessary configurations have been completed to ensure that all MCU components will be updated on WRMSR 79H.	
3	ARCH_ROLLBACK_SVN_COMMIT When set to 1, indicates support for the MCU deferred SVN architecture, SVN reporting architecture, and MCU rollback architecture.	
4	MCU_STAGING When set to 1, indicates that the microcode update staging capability is supported by the processor. When supported, the use of the MCU staging capability is recommended to reduce the latency of the IA32_BIOS_UPDT_TRIG operation.	
7:5	Reserved for future use.	
15:8	UNIFORM_MCU_SCOPE Indicates the current* uniform microcode update scope: ▪ 0x02: Core Scoped ▪ 0x03: Module Scoped** ▪ 0x04: Tile Scoped** ▪ 0x05: Die Scoped** ▪ 0x80: Package Scoped ▪ 0xC0: Platform Scoped All others: Reserved for future use * The value of this field reflects the state of platform configuration and may change as the configuration changes during the boot process. Once configuration is complete, it is not expected to change during runtime. ** If these domains are enumerated by CPUID.1F, then this field may also report them as appropriate.	
63:16	Reserved for future use.	
Register Address: 7CH, 124		IA32_MCU_STATUS

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
MCU Status (R/O) Communicates results from the previous patch loads.		
0	MCU_PARTIAL_UPDATE When set to 1, indicates that the most recent write to IA32_BIOS_UPDT_TRIG resulted in a partial update. This means that microcode update components were only partially updated after some portion of the MCU had already been committed and the Revision ID had been updated.	
1	AUTH_FAIL_ON_MCU_COMPONENT When set to 1, indicates that an authentication failure occurred on some portion of the MCU after another portion of the MCU had already been committed and the Revision ID had already been updated on the most recent write to IA32_BIOS_UPDT_TRIG.	
2	Reserved for future use.	
3	POST_BIOS_MCU When set to 1, indicates that an update was successfully loaded via IA32_BIOS_UPDT_TRIG after bit 0 of MSR_BIOS_DONE (address 151H) was set to 1.	
63:4	Reserved for future use.	
Register Address: 82H, 130		IA32_FZM_RANGE_INDEX
IA32_FZM_RANGE_INDEX (R/W) Index and Domain handle for a valid FZM region. Programmed by software and used by other FRM MSRs FZM Range Index register to R/W Domain Index.		
3:0	REGION_INDEX Holds the Index of domain.	
7:4	Reserved.	
12:8	DOMAIN_HANDLE Holds the Domain Handle.	
63:13	Reserved.	
Register Address: 83H, 131		IA32_FZM_DOMAIN_CONFIG
IA32_FZM_DOMAIN_CONFIG (R/O) Bit mask of valid regions within the domain identified by FZM_RANGE_INDEX.		
63:0	REGION_BITMAP Bitmap of valid regions for a given domain.	
Register Address: 84H, 132		IA32_FZM_RANGE_STARTADDR
IA32_FZM_RANGE_STARTADDR (R/O) Start address of the FZM range pointed to by FZM_RANGE_INDEX.		
51:0	START_ADDR Start address of the specified domain in FZM_RANGE_INDEX.	
63:52	Reserved.	
Register Address: 85H, 133		IA32_FZM_RANGE_ENDADDR

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
IA32_FZM_RANGE_ENDADDR (R/O) End address of the specified domain in FZM_RANGE_INDEX.		
51:0	END_ADDR End address of the specified domain in FZM_RANGE_INDEX.	
63:52	Reserved.	
Register Address: 86H, 134	IA32_FZM_RANGE_WRITESTATUS	
IA32_FZM_RANGE_WRITESTATUS (R/O) Write status of the FZM range pointed to by FZM_RANGE_INDEX.		
0	WRITE_STATUS Write status of the specified domain in FZM_RANGE_INDEX.	
1	READ_STATUS Read status of the specified domain in FZM_RANGE_INDEX.	
63:2	Reserved.	
Register Address: 87H, 135	IA32_MKTME_KEYID_PARTITIONING	
MKTME KEY ID Partitioning (R/O) Enumerates the number of activated KeyIDs for Intel TME-MK and Intel TDX.		
31:0	NUM_MKTME_KIDS Number of activated Intel TME-MK KeyIDs. This field is supported on all parts that enumerate support for Intel Total Memory Encryption - Multi-Key (Intel TME-MK). If IA32_TME_ACTIVATE.LOCK is 1, this field reports MAX_ACTIVATE_MKTME_HKIDS (KMK-1) else report 0. Intel TME-MK KIDs will always span the KID range [1 ... NUM_MKTME_KIDS].	
63:32	NUM_TDX_PRIV_KIDS Number of activated TDX private KeyIDs. This field is supported on all parts that enumerate support for SEAM mode. If IA32_TME_ACTIVATE.LOCK is 1, This field reports MAX_ACTIVATE_TDX_HKIDS (KTD) else report 0. TDX private KIDs will always span the range [NUM_MKTME_KIDS+1... (NUM_MKTME_KIDS + NUM_TDX_PRIV_KIDS)].	
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID (BIOS_SIGN/BBL_CR_D3)	
BIOS Update Signature (R/W) Returns the microcode update signature following the execution of CPUID.01H. A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.		06_01H
31:0	Reserved.	
63:32	PATCH_SIGN_ID It is recommended that this field be preloaded with zero prior to executing CPUID. If the field remains zero following the execution of CPUID, this indicates that no microcode update is loaded. Any non-zero value is the microcode update signature patch signature ID.	
Register Address: 8CH, 140	IA32_SGXLEPUBKEYHASH0	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
IA32_SGXLEPUBKEYHASH[63:0] (R/W) Bits 63:0 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.		Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && CPUID.(EAX=07H, ECX=0H):ECX[30]=1. Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1.
Register Address: 8DH, 141		IA32_SGXLEPUBKEYHASH1
IA32_SGXLEPUBKEYHASH[127:64] (R/W) Bits 127:64 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.		Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.
Register Address: 8EH, 142		IA32_SGXLEPUBKEYHASH2
IA32_SGXLEPUBKEYHASH[191:128] (R/W) Bits 191:128 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.		Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.
Register Address: 8FH, 143		IA32_SGXLEPUBKEYHASH3
IA32_SGXLEPUBKEYHASH[255:192] (R/W) Bits 255:192 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.		Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.
Register Address: 90H, 144		IA32_SGXLEPUBKEYHASH4
IA32_SGXLEPUBKEYHASH[319:256] (R/W) Bits 319:256 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.		Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.
Register Address: 91H, 145		IA32_SGXLEPUBKEYHASH5
IA32_SGXLEPUBKEYHASH[383:320] (R/W) Bits 383:320 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.		Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.
Register Address: 9BH, 155		IA32_SMM_MONITOR_CTL
SMM Monitor Configuration (R/W)		If CPUID.01H: ECX[5]=1 || CPUID.01H: ECX[6] = 1
0	Valid (R/W)	
1	Reserved.	
2	Controls SMI unblocking by VMXOFF (see Section 33.14.4).	If IA32_VMX_MISC[28]
11:3	Reserved.	
31:12	MSEG Base (R/W)	
63:32	Reserved.	
Register Address: 9EH, 158		IA32_SMBASE
Base address of the logical processor's SMRAM image (R/O, SMM only).		If IA32_VMX_MISC[15]
Register Address: BCH, 188		IA32_MISC_PACKAGE_CTLS

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Power Filtering Control (R/W) This MSR has a value of 0 after reset and is unaffected by INIT# or SIPI#.		If IA32_ARCH_CAPABILITIES [10] = 1
0	ENERGY_FILTERING_ENABLE (R/W) If set, RAPL MSRs report filtered processor power consumption data. This bit can be changed from 0 to 1, but cannot be changed from 1 to 0. After setting, all attempts to clear it are ignored until the next processor reset.	If IA32_ARCH_CAPABILITIES [11] = 1
63:1	Reserved.	
Register Address: BDH, 189	IA32_XAPIC_DISABLE_STATUS	
xAPIC Disable Status (R/O)		If CPUID.(EAX-07H, ECX=0):EDX[29]=1 and IA32_ARCH_CAPABILITIES [21] = 1
0	LEGACY_XAPIC_DISABLED When set, indicates that the local APIC is in x2APIC mode (IA32_APIC_BASE.EXTD = 1) and that attempts to clear IA32_APIC_BASE.EXTD will fail (e.g., WRMSR will #GP).	
63:1	Reserved.	
Register Address: C1H, 193	IA32_PMC0 (PERFCTR0)	
General Performance Counter 0 (R/W)		If CPUID.0AH: EAX[15:8] > 0
Register Address: C2H, 194	IA32_PMC1 (PERFCTR1)	
General Performance Counter 1 (R/W)		If CPUID.0AH: EAX[15:8] > 1
Register Address: C3H, 195	IA32_PMC2	
General Performance Counter 2 (R/W)		If CPUID.0AH: EAX[15:8] > 2
Register Address: C4H, 196	IA32_PMC3	
General Performance Counter 3 (R/W)		If CPUID.0AH: EAX[15:8] > 3
Register Address: C5H, 197	IA32_PMC4	
General Performance Counter 4 (R/W)		If CPUID.0AH: EAX[15:8] > 4
Register Address: C6H, 198	IA32_PMC5	
General Performance Counter 5 (R/W)		If CPUID.0AH: EAX[15:8] > 5
Register Address: C7H, 199	IA32_PMC6	
General Performance Counter 6 (R/W)		If CPUID.0AH: EAX[15:8] > 6
Register Address: C8H, 200	IA32_PMC7	
General Performance Counter 7 (R/W)		If CPUID.0AH: EAX[15:8] > 7
Register Address: C9H, 201	IA32_PMC8	
General Performance Counter 8 (R/W)		If CPUID.0AH: EAX[15:8] > 8
Register Address: CAH, 202	IA32_PMC9	
General Performance Counter 9 (R/W)		If CPUID.0AH: EAX[15:8] > 9
Register Address: CFH, 207	IA32_CORE_CAPABILITIES	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
IA32 Core Capabilities Register		If CPUID.(EAX=07H, ECX=0):EDX[30] = 1
63:0	Reserved.	No architecturally defined bits.
Register Address: E1H, 225	IA32_UMWAIT_CONTROL	
UMWAIT Control (R/W)		
0	C0.2 is not allowed by the OS. Value of “1” means all C0.2 requests revert to C0.1.	
1	Reserved.	
31:2	Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.	
Register Address: E7H, 231	IA32_MPERF	
TSC Frequency Clock Counter (R/Write to clear)		If CPUID.06H: ECX[0] = 1
63:0	C0_MCNT: C0 TSC Frequency Clock Count Increments at fixed interval (relative to TSC freq.) when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_APERF.	
Register Address: E8H, 232	IA32_APERF	
Actual Performance Clock Counter (R/Write to clear)		If CPUID.06H: ECX[0] = 1
63:0	C0_ACNT: C0 Actual Frequency Clock Count Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_MPERF.	
Register Address: FEH, 254	IA32_MTRRCAP (MTRRcap)	
MTRR Capability (R/O) See Section 13.11.2.1, “IA32_MTRR_DEF_TYPE MSR.”		06_01H
7:0	VCNT: The number of variable memory type ranges in the processor.	
8	Fixed range MTRRs are supported when set.	
9	Reserved.	
10	WC Supported when set.	
11	SMRR Supported when set.	
12	PRMRR supported when set.	
63:13	Reserved.	
Register Address: 10AH, 266	IA32_ARCH_CAPABILITIES	
Enumeration of Architectural Features (R/O)		If CPUID.(EAX=07H, ECX=0):EDX[29]=1
0	RDCL_NO: The processor is not susceptible to Rogue Data Cache Load (RDCL).	
1	IBRS_ALL: The processor supports enhanced IBRS.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	MSR/Bit Description	Comment
2	RSBA: The processor supports RSB Alternate. Alternative branch predictors may be used by RET instructions when the RSB is empty. SW using retpoline may be affected by this behavior.	
3	SKIP_L1DFL_VMENTRY: A value of 1 indicates the hypervisor need not flush the L1D on VM entry.	
4	SSB_NO: Processor is not susceptible to Speculative Store Bypass.	
5	MDS_NO: Processor is not susceptible to Microarchitectural Data Sampling (MDS).	
6	IF_PSCHANGE_MC_NO: The processor is not susceptible to a machine check error due to modifying the size of a code page without TLB invalidation.	
7	TSX_CTRL: If 1, indicates presence of IA32_TSX_CTRL MSR.	
8	TAA_NO: If 1, processor is not affected by TAA.	
9	MCU_CONTROL: If 1, the processor supports the IA32_MCU_CONTROL MSR.	
10	MISC_PACKAGE_CTLS: The processor supports IA32_MISC_PACKAGE_CTLS MSR.	
11	ENERGY_FILTERING_CTL: The processor supports setting and reading the IA32_MISC_PACKAGE_CTLS[0] (ENERGY_FILTERING_ENABLE) bit.	
12	DOITM: If 1, the processor supports Data Operand Independent Timing Mode.	
13	SBDR_SSDP_NO: The processor is not affected by either the Shared Buffers Data Read (SBDR) vulnerability or the Sideband Stale Data Propagator (SSDP).	
14	FBSDP_NO: The processor is not affected by the Fill Buffer Stale Data Propagator (FBSDP).	
15	PSDP_NO: The processor is not affected by vulnerabilities involving the Primary Stale Data Propagator (PSDP).	
16	MCU_ENUMERATION: If 1, the processor supports the IA32_MCU_ENUMERATION and IA32_MCU_STATUS MSRs.	
17	FB_CLEAR: If 1, the processor supports overwrite of fill buffer values as part of MD_CLEAR operations with the VERW instruction.	
18	FB_CLEAR_CTRL: If 1, the processor supports the IA32_MCU_OPT_CTRL MSR and allows software to set bit 3 of that MSR (FB_CLEAR_DIS).	
19	RRSBA: A value of 1 indicates the processor may have the RRSBA alternate prediction behavior, if not disabled by RRSBA_DIS_U or RRSBA_DIS_S.	
20	BHI_NO: A value of 1 indicates BHI_NO branch prediction behavior, regardless of the value of IA32_SPEC_CTRL[BHI_DIS_S] MSR bit.	
21	XAPIC_DISABLE_STATUS: Enumerates that the IA32_XAPIC_DISABLE_STATUS MSR exists, and that bit 0 specifies whether the legacy xAPIC is disabled and APIC state is locked to x2APIC.	
22	MCU_EXTENDED_SERVICE: If 1, the processor supports MCU Extended servicing - IA32_MCU_EXT_SERVICE MSR.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
23	OVERCLOCKING_STATUS: If set, the IA32_OVERCLOCKING_STATUS MSR exists.	
24	PBRSB_NO: If 1, the processor is not affected by issues related to Post-Barrier Return Stack Buffer Predictions.	
25	GDS_CTRL: If 1, the processor supports the GDS_MITG_DIS and GDS_MITG_LOCK bits of the IA32_MCU_OPT_CTRL MSR.	
26	GDS_NO: If 1, the processor is not affected by Gather Data Sampling.	
27	RFDS_NO: If 1, the processor is not affected by Register File Data Sampling.	
28	RFDS_CLEAR: If 1, when VERW is executed the processor will clear stale data from register files affected by Register File Data Sampling.	
29	IGN_UMONITOR_SUPPORT If 0, IA32_MCU_OPT_CTRL bit 6 (IGN_UMONITOR) is not supported. If 1, it indicates support of IA32_MCU_OPT_CTRL bit 6 (IGN_UMONITOR).	
30	MON_UMON_MITG_SUPPORT If 0, IA32_MCU_OPT_CTRL bit 7 (MON_UMON_MITG) is not supported. If 1, it indicates support of IA32_MCU_OPT_CTRL bit 7 (MON_UMON_MITG).	
63:31	Reserved.	
Register Address: 10BH, 267	IA32_FLUSH_CMD	
Flush Command (WO) Gives software a way to invalidate structures with finer granularity than other architectural methods.		If any one of the enumeration conditions for defined bit field positions holds.
0	L1D_FLUSH Writeback and invalidate the L1 data cache.	If CPUID.(EAX=07H, ECX=0):EDX[28]=1
63:1	Reserved.	
Register Address: 10FH, 271	IA32_TSX_FORCE_ABORT	
TSX Force Abort		If CPUID.(EAX=07H, ECX=0):EDX[13]=1
0	RTM_FORCE_ABORT If 1, all RTM transactions abort with EAX code 0.	R/W, Default: 0 If CPUID.(EAX=07H,ECX=0): EDX[11]=1, bit 0 is always 1 and writes to change it are ignored. If SDV_ENABLE_RTM is 1, bit 0 is always 0 and writes to change it are ignored.
1	TSX_CPUID_CLEAR When set, CPUID.(EAX=07H,ECX=0):EBX[11]=0 and CPUID.(EAX=07H,ECX=0):EBX[4]=0.	R/W, Default: 0 Can be set only if CPUID.(EAX=07H,ECX=0): EDX[11]=1 or if SDV_ENABLE_RTM is 1.

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
2	SDV_ENABLE_RTM When set, CPUID.(EAX=07H,ECX=0):EDX[11]=0 and the processor may not force abort RTM. This unsupported mode should only be used for software development and not for production usage.	R/W, Default: 0 If 0, can be set only if CPUID.(EAX=07H,ECX=0): EDX[11]=1.
63:3	Reserved.	
Register Address: 122H, 290		IA32_TSX_CTRL
IA32_TSX_CTRL (R/W)		Thread scope. Not architecturally serializing. Available when CPUID.ARCH_CAP(EAX=7H, ECX = 0):EDX[29] = 1 and IA32_ARCH_CAPABILITIES.bit 7 = 1.
0	RTM_DISABLE When set to 1, XBEGIN will always abort with EAX code 0.	
1	TSX_CPUID_CLEAR When set to 1, CPUID.07H.EBX.RTM [bit 11] and CPUID.07H.EBX.HLE [bit 4] report 0. When set to 0 and the SKU supports TSX, these bits will return 1.	
63:2	Reserved.	
Register Address: 123H, 291		IA32_MCU_OPT_CTRL
Microcode Update Option Control (R/W)		If CPUID.(EAX=07H, ECX=0):EDX[9]=1 or CPUID.(EAX=07H, ECX=0H):EDX[11]=1 IA32_ARCH_CAPABILITIES [18] = 1 or IA32_ARCH_CAPABILITIES [25]=1 or IA32_ARCH_CAPABILITIES [29]=1 or IA32_ARCH_CAPABILITIES [30]=1
0	RNGDS_MITG_DIS (R/W) If 0 (default), SRBDS mitigation is enabled for RDRAND and RDSEED. If 1, SRBDS mitigation is disabled for RDRAND and RDSEED executed outside of Intel SGX enclaves.	If CPUID.(EAX=07H, ECX=0):EDX[9]=1
1	RTM_ALLOW If 0, XBEGIN will always abort with EAX code 0. If 1, XBEGIN behavior depends on the value of IA32_TSX_CTRL[RTM_DISABLE].	If CPUID.(EAX=07H, ECX=0H):EDX[11]=1 Read/Write Setting RTM_LOCKED prevents writes to this bit.
2	RTM_LOCKED When 1, RTM_ALLOW is locked at zero, writes to RTM_ALLOW will be ignored.	If CPUID.(EAX=07H, ECX=0H):EDX[11]=1 Read-Only status bit.
3	FB_CLEAR_DIS If 1, prevents the VERW instruction from performing an FB_CLEAR action.	If IA32_ARCH_CAPABILITIES [18]=1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
4	GDS_MITG_DIS If 0, the Gather Data Sampling mitigation is enabled (patch load time default). If 1 on all threads for a given core, the Gather Data Sampling mitigation is disabled.	If IA32_ARCH_CAPABILITIES [25]=1
5	GDS_MITG_LOCK If 0, not locked, and GDS_MITG_DIS is under OS control. If 1, locked and GDS_MITG_DIS is forced to 0 (writes are ignored).	If IA32_ARCH_CAPABILITIES [25]=1
6	IGN_UMONITOR If 0, enable CPL0-3 software to use the UMONITOR/UMWAIT instructions. If 1 (default), disable UMONITOR functionality. CPL0-3 software will be able to call the UMONITOR instruction without causing a fault, however the address monitoring hardware will not be armed. When UMWAIT is called, it will not enter an implementation-dependent optimized state.	If IA32_ARCH_CAPABILITIES [29]=1
7	MON_UMON_MITG If 0 (default), disabled. If 1, enable: Flush the thread's previously monitored address from the CPU caches as part of the (U)MONITOR instruction. Additionally, for every 4th (U)MONITOR instruction within a core, flush the peer hyperthread's monitored address from the CPU caches as well. This will increase the latency of the instruction. This may have a minor impact on workloads using the (U)MONITOR instruction.	If IA32_ARCH_CAPABILITIES [30]=1
63:8	Reserved.	
Register Address: 174H, 372		IA32_SYSENTER_CS
SYSENTER_CS_MSR (R/W)		06_01H
15:0	CS Selector.	
31:16	Not used.	Can be read and written.
63:32	Not used.	Writes ignored; reads return zero.
Register Address: 175H, 373		IA32_SYSENTER_ESP
SYSENTER_ESP_MSR (R/W)		06_01H
Register Address: 176H, 374		IA32_SYSENTER_EIP
SYSENTER_EIP_MSR (R/W)		06_01H
Register Address: 179H, 377		IA32_MCG_CAP (MCG_CAP)
Global Machine Check Capability (R/O)		06_01H
7:0	Count: Number of reporting banks.	
8	MCG_CTL_P: IA32_MCG_CTL is present if this bit is set.	
9	MCG_EXT_P: Extended machine check state registers are present if this bit is set.	
10	MCP_CMCI_P: Support for corrected MC error event is present.	06_01H
11	MCG_TES_P: Threshold-based error status register are present if this bit is set.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
15:12	Reserved.	
23:16	MCG_EXT_CNT: Number of extended machine check state registers present.	
24	MCG_SER_P: The processor supports software error recovery if this bit is set.	
25	Reserved.	
26	MCG_ELOG_P: Indicates that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format “Generic Error Data Entry” that augments the data included in machine check bank registers.	06_3EH
27	MCG_LMCE_P: Indicates that the processor supports extended state in IA32_MCG_STATUS and associated MSR necessary to configure Local Machine Check Exception (LMCE).	06_3EH
63:28	Reserved.	
Register Address: 17AH, 378		IA32_MCG_STATUS (MCG_STATUS)
Global Machine Check Status (R/W)		06_01H
0	RIPV. Restart IP valid.	06_01H
1	EIPV. Error IP valid.	06_01H
2	MCIP. Machine check in progress.	06_01H
3	LMCE_S.	If IA32_MCG_CAP.LMCE_P[27] =1
63:4	Reserved.	
Register Address: 17BH, 379		IA32_MCG_CTL (MCG_CTL)
Global Machine Check Control (R/W)		If IA32_MCG_CAP.CTL_P[8] =1
Register Address: 180H—185H, 384—389		N/A
Reserved		06_0EH[2]
Register Address: 186H, 390		IA32_PERFEVTSEL0 (PERFEVTSEL0)
Performance Event Select Register 0 (R/W)		If CPUID.0AH: EAX[15:8] > 0
7:0	Event Select: Selects a performance event logic unit.	
15:8	UMask: Qualifies the microarchitectural condition to detect on the selected event logic.	
16	USR: Counts while in privilege level is not ring 0.	
17	OS: Counts while in privilege level is ring 0.	
18	Edge: Enables edge detection if set.	
19	PC: Enables pin control.	
20	INT: Enables interrupt on counter overflow.	
21	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	
22	EN: Enables the corresponding performance counter to commence counting when this bit is set.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
23	INV: Invert the CMASK.	
31:24	CMASK: When CMASK is not zero, the corresponding performance counter increments each cycle if the event count is greater than or equal to the CMASK.	
63:32	Reserved.	
Register Address: 187H, 391	IA32_PERFEVTSEL1 (PERFEVTSEL1)	
Performance Event Select Register 1 (R/W)		If CPUID.0AH: EAX[15:8] > 1
Register Address: 188H, 392	IA32_PERFEVTSEL2	
Performance Event Select Register 2 (R/W)		If CPUID.0AH: EAX[15:8] > 2
Register Address: 189H, 393	IA32_PERFEVTSEL3	
Performance Event Select Register 3 (R/W)		If CPUID.0AH: EAX[15:8] > 3
Register Address: 18AH, 394	IA32_PERFEVTSEL4	
Performance Event Select Register 4 (R/W)		If CPUID.0AH: EAX[15:8] > 4
Register Address: 18BH, 395	IA32_PERFEVTSEL5	
Performance Event Select Register 5 (R/W)		If CPUID.0AH: EAX[15:8] > 5
Register Address: 18CH, 396	IA32_PERFEVTSEL6	
Performance Event Select Register 6 (R/W)		If CPUID.0AH: EAX[15:8] > 6
Register Address: 18DH, 397	IA32_PERFEVTSEL7	
Performance Event Select Register 7 (R/W)		If CPUID.0AH: EAX[15:8] > 7
Register Address: 18EH, 398	IA32_PERFEVTSEL8	
Performance Event Select Register 8 (R/W)		If CPUID.0AH: EAX[15:8] > 8
Register Address: 18FH, 399	IA32_PERFEVTSEL9	
Performance Event Select Register 9 (R/W)		If CPUID.0AH: EAX[15:8] > 9
Register Address: 18AH—194H, 394—404	N/A	
Reserved.		06_0EH[3]
Register Address: 195H, 405	IA32_OVERCLOCKING_STATUS	
Overclocking Status (R/O) IA32_ARCH_CAPABILITIES[bit 23] enumerates support for this MSR.		
0	Overclocking Utilized Indicates if specific forms of overclocking have been enabled on this boot or reset cycle: 0 indicates no, 1 indicates yes.	
1	Undervolt Protection Indicates if the “Dynamic OC Undervolt Protection” security feature is active: 0 indicates disabled, 1indicates enabled.	
2	Overclocking Secure Status Indicates that overclocking capabilities have been unlocked by BIOS, with or without overclocking: 0 indicates Not Secured, 1 indicates Secure.	
63:3	Reserved.	
Register Address: 196H—197H, 406—407	N/A	
Reserved.		06_0EH[3]

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 198H, 408	IA32_PERF_STATUS	
Current Performance Status (R/O) See Section 16.1.1, "Software Interface For Initiating Performance State Transitions."		0F_03H
15:0	Current Performance State Value.	
63:16	Reserved.	
Register Address: 199H, 409	IA32_PERF_CTL	
Performance Control MSR (R/W) Software makes a request for a new Performance state (P-State) by writing this MSR. See Section 16.1.1, "Software Interface For Initiating Performance State Transitions."		0F_03H
15:0	Target performance State Value.	
31:16	Reserved.	
32	Intel® Dynamic Acceleration Technology Engage (R/W) When set to 1: Disengages Intel Dynamic Acceleration Technology.	06_0FH (Mobile only)
63:33	Reserved.	
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation Control (R/W) See Section 16.8.3, "Software Controlled Clock Modulation."		If CPUID.01H:EDX[22] = 1
0	Extended On-Demand Clock Modulation Duty Cycle.	If CPUID.06H:EAX[5] = 1
3:1	On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation.	If CPUID.01H:EDX[22] = 1
4	On-Demand Clock Modulation Enable: Set 1 to enable modulation.	If CPUID.01H:EDX[22] = 1
63:5	Reserved.	
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the processor's thermal sensors and thermal monitor. See Section 16.8.2, "Thermal Monitor."		If CPUID.01H:EDX[22] = 1
0	High-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
1	Low-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
2	PROCHOT# Interrupt Enable	If CPUID.01H:EDX[22] = 1
3	FORCEPR# Interrupt Enable	If CPUID.01H:EDX[22] = 1
4	Critical Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
7:5	Reserved.	
14:8	Threshold #1 Value	If CPUID.01H:EDX[22] = 1
15	Threshold #1 Interrupt Enable	If CPUID.01H:EDX[22] = 1
22:16	Threshold #2 Value	If CPUID.01H:EDX[22] = 1
23	Threshold #2 Interrupt Enable	If CPUID.01H:EDX[22] = 1
24	Power Limit Notification Enable	If CPUID.06H:EAX[4] = 1
25	Hardware Feedback Notification Enable	If CPUID.06H:EAX[24] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63:26	Reserved.	
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Status Information (R/O) Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. See Section 16.8.2, "Thermal Monitor."		If CPUID.01H:EDX[22] = 1
0	Thermal Status (R/O)	If CPUID.01H:EDX[22] = 1
1	Thermal Status Log (R/W)	If CPUID.01H:EDX[22] = 1
2	PROCHOT # or FORCEPR# event (R/O)	If CPUID.01H:EDX[22] = 1
3	PROCHOT # or FORCEPR# log (R/WC0)	If CPUID.01H:EDX[22] = 1
4	Critical Temperature Status (R/O)	If CPUID.01H:EDX[22] = 1
5	Critical Temperature Status log (R/WC0)	If CPUID.01H:EDX[22] = 1
6	Thermal Threshold #1 Status (R/O)	If CPUID.01H:ECX[8] = 1
7	Thermal Threshold #1 log (R/WC0)	If CPUID.01H:ECX[8] = 1
8	Thermal Threshold #2 Status (R/O)	If CPUID.01H:ECX[8] = 1
9	Thermal Threshold #2 log (R/WC0)	If CPUID.01H:ECX[8] = 1
10	Power Limitation Status (R/O)	If CPUID.06H:EAX[4] = 1
11	Power Limitation log (R/WC0)	If CPUID.06H:EAX[4] = 1
12	Current Limit Status (R/O)	If CPUID.06H:EAX[7] = 1
13	Current Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
14	Cross Domain Limit Status (R/O)	If CPUID.06H:EAX[7] = 1
15	Cross Domain Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
22:16	Digital Readout (R/O)	If CPUID.06H:EAX[0] = 1
26:23	Reserved.	
30:27	Resolution in Degrees Celsius (R/O)	If CPUID.06H:EAX[0] = 1
31	Reading Valid (R/O)	If CPUID.06H:EAX[0] = 1
63:32	Reserved.	
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		
0	Fast-Strings Enable When set, the fast-strings feature (for REP MOVS and REP STORS) is enabled (default). When clear, fast-strings are disabled.	0F_0H
2:1	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
3	Automatic Thermal Control Circuit Enable (R/W) 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation. 0 = Disabled. Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2, and adaptive thermal throttling will still be activated. The default value of this field varies with product. See respective tables where default value is listed.	0F_0H
6:4	Reserved.	
7	Performance Monitoring Available (R) 1 = Performance monitoring enabled. 0 = Performance monitoring disabled.	0F_0H
10:8	Reserved.	
11	Branch Trace Storage Unavailable (R/O) 1 = Processor doesn't support branch trace storage (BTS). 0 = BTS is supported.	0F_0H
12	Processor Event Based Sampling (PEBS) Unavailable (R/O) 1 = PEBS is not supported. 0 = PEBS is supported.	06_0FH
15:13	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) 0= Enhanced Intel SpeedStep Technology disabled. 1 = Enhanced Intel SpeedStep Technology enabled.	If CPUID.01H: ECX[7] =1
17	Reserved.	
18	ENABLE MONITOR FSM (R/W) When this bit is set to 0, the MONITOR feature flag is not set (CPUID.01H:ECX[bit 3] = 0). This indicates that MONITOR/MWAIT are not supported. Software attempts to execute MONITOR/MWAIT will cause #UD when this bit is 0. When this bit is set to 1 (default), MONITOR/MWAIT are supported (CPUID.01H:ECX[bit 3] = 1). If the SSE3 feature flag ECX[0] is not set (CPUID.01H:ECX[bit 0] = 0), the OS must not attempt to alter this bit. BIOS must leave it in the default state. Writing this bit when the SSE3 feature flag is set to 0 may generate a #GP exception.	0F_03H
21:19	Reserved.	
22	Limit CPUID Maxval (R/W) When this bit is set to 1, CPUID.00H returns a maximum value in EAX[7:0] of 2.	CPUID.0H:EAX > 2 and CPUID.(EAX = 07H, ECX = 1):EBX.CPUIDMAXVAL_LIM_RMV [bit 3] = 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
23	xTPR Message Disable (R/W) When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority.	If CPUID.01H:ECX[14] = 1
63:24	Reserved. Note: Some older processors defined one of these bits as a disable for the execute-disable feature of paging. If a processor supports this bit, this information is provided in the model-specific tables. See Table 2-3 for the definition of this bit.	
Register Address: 1B0H, 432	IA32_ENERGY_PERF_BIAS	
Performance Energy Bias Hint (R/W)		If CPUID.6H:ECX[3] = 1
3:0	Power Policy Preference: 0 indicates preference to highest performance. 15 indicates preference to maximize energy saving.	
63:4	Reserved.	
Register Address: 1B1H, 433	IA32_PACKAGE_THERM_STATUS	
Package Thermal Status Information (R/O) Contains status information about the package's thermal sensor. See Section 16.9, "Package Level Thermal Management."		If CPUID.06H: EAX[6] = 1
0	Pkg Thermal Status (R/O)	
1	Pkg Thermal Status Log (R/W)	
2	Pkg PROCHOT # event. (R/O)	
3	Pkg PROCHOT # log. (R/WC0)	
4	Pkg Critical Temperature Status. (R/O)	
5	Pkg Critical Temperature Status Log. (R/WC0)	
6	Pkg Thermal Threshold #1 Status. (R/O)	
7	Pkg Thermal Threshold #1 Log. (R/WC0)	
8	Pkg Thermal Threshold #2 Status. (R/O)	
9	Pkg Thermal Threshold #1 Log. (R/WC0)	
10	Pkg Power Limitation Status. (R/O)	
11	Pkg Power Limitation Log. (R/WC0)	
15:12	Reserved.	
22:16	Pkg Digital Readout. (R/O)	
25:23	Reserved.	
26	Hardware Feedback Interface Structure Change Status.	If CPUID.06H:EAX.[19] = 1
63:27	Reserved.	
Register Address: 1B2H, 434	IA32_PACKAGE_THERM_INTERRUPT	
Pkg Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the package's thermal sensor. See Section 16.9, "Package Level Thermal Management."		If CPUID.06H: EAX[6] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
0	Pkg High-Temperature Interrupt Enable.	
1	Pkg Low-Temperature Interrupt Enable.	
2	Pkg PROCHOT# Interrupt Enable.	
3	Reserved.	
4	Pkg Overheat Interrupt Enable.	
7:5	Reserved.	
14:8	Pkg Threshold #1 Value.	
15	Pkg Threshold #1 Interrupt Enable.	
22:16	Pkg Threshold #2 Value.	
23	Pkg Threshold #2 Interrupt Enable.	
24	Pkg Power Limit Notification Enable.	
25	Hardware Feedback Interrupt Enable.	If CPUID.06H:EAX.[19] = 1
63:26	Reserved.	
Register Address: 1C4H, 452		IA32_XFD

Extended Feature Disable Control (R/W)
Controls which XSAVE-enabled features are temporarily disabled.
See Section 13.14 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
		If CPUID.(EAX=0DH,ECX=1):
		EAX[4] = 1

Register Address: 1C5H, 453		IA32_XFD_ERR

Extended Feature Disable Error Code (R/W)
Reports which XSAVE-enabled features caused a fault due to being disabled.
See Section 13.14 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
		If CPUID.(EAX=0DH,ECX=1):
		EAX[4] = 1

Register Address: 1D9H, 473		IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)
Trace/Profile Resource Control (R/W)		06_0EH
0	LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	06_01H
1	BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	06_01H
2	BLD: Enable OS bus-lock detection. See Section 19.3.1.6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.	If (CPUID.(EAX=07H, ECX=0):ECX[24] = 1)
5:3	Reserved.	
6	TR: Setting this bit to 1 enables branch trace messages to be sent.	06_0EH
7	BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	06_0EH
8	BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	06_0EH
9	1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.	06_0FH
10	BTS_OFF_USR: When set, BTS or BTM is skipped if CPL > 0.	06_0FH

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
11	FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
12	FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 38FH) on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
13	ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.	06_1AH
14	FREEZE_WHILE_SMM: When set, freezes PerfMon and trace messages while in SMM.	If IA32_PERF_CAPABILITIES[12] = 1
15	RTM_DEBUG: When set, enables DR7 debug bit on XBEGIN.	If (CPUID.(EAX=07H, ECX=0):EBX[11] = 1)
63:16	Reserved.	
Register Address: 1DDH, 477	IA32_LER_FROM_IP	
Last Event Record Source IP Register (R/W)		
63:0	FROM_IP The source IP of the recorded branch or event, in canonical form.	Reset Value: 0
Register Address: 1DEH, 478	IA32_LER_TO_IP	
Last Event Record Destination IP Register (R/W)		
63:0	TO_IP The destination IP of the recorded branch or event, in canonical form.	Reset Value: 0
Register Address: 1E0H, 480	IA32_LER_INFO	
Last Event Record Info Register (R/W)		
55:0	Undefined, may be zero or non-zero. Writes of non- zero values do not fault, but reads may return a different value.	Reset Value: 0
59:56	BR_TYPE The branch type recorded by this LBR. Encodings match those of IA32_LBR_x_INFO.	Reset Value: 0
60	Undefined, may be zero or non-zero. Writes of non- zero values do not fault, but reads may return a different value.	Reset Value: 0
61	TSX_ABORT This LBR record is a TSX abort. On processors that do not support Intel® TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.	Reset Value: 0
62	IN_TSX This LBR record records a branch that retired during a TSX transaction. On processors that do not support Intel® TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.	Reset Value: 0
63	MISPRED The recorded branch taken/not-taken resolution (for conditional branches) or target (for any indirect branch, including RETs) was mispredicted.	Reset Value: 0
Register Address: 1F2H, 498	IA32_SMRR_PHYSBASE	
SMRR Base Address (Writeable only in SMM) Base address of SMM memory range.		If IA32_MTRRCAP.SMRR[11] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
7:0	Type. Specifies memory type of the range.	
11:8	Reserved.	
31:12	PhysBase SMRR physical Base Address.	
63:32	Reserved.	
Register Address: 1F3H, 499	IA32_SMRR_PHYSMASK	
SMRR Range Mask (Writeable only in SMM) Range Mask of SMM memory range.		If IA32_MTRRCAP[SMRR] = 1
10:0	Reserved.	
11	Valid Enable range mask.	
31:12	PhysMask SMRR address range mask.	
63:32	Reserved.	
Register Address: 1F8H, 504	IA32_PLATFORM_DCA_CAP	
DCA Capability (R)		If CPUID.01H: ECX[18] = 1
Register Address: 1F9H, 505	IA32_CPU_DCA_CAP	
If set, CPU supports Prefetch-Hint type.		If CPUID.01H: ECX[18] = 1
Register Address: 1FAH, 506	IA32_DCA_0_CAP	
DCA type 0 Status and Control register.		If CPUID.01H: ECX[18] = 1
0	DCA_ACTIVE: Set by HW when DCA is fuse-enabled and no defeatures are set.	
2:1	TRANSACTION	
6:3	DCA_TYPE	
10:7	DCA_QUEUE_SIZE	
12:11	Reserved.	
16:13	DCA_DELAY: Writes will update the register but have no HW side-effect.	
23:17	Reserved.	
24	SW_BLOCK: SW can request DCA block by setting this bit.	
25	Reserved.	
26	HW_BLOCK: Set when DCA is blocked by HW (e.g., CR0.CD = 1).	
31:27	Reserved.	
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0 (MTRRphysBase0)	
See Section 13.11.2.3, "Variable Range MTRRs."		If IA32_MTRRCAP[7:0] > 0
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0	
MTRRphysMask0		If IA32_MTRRCAP[7:0] > 0
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1	
MTRRphysBase1		If IA32_MTRRCAP[7:0] > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1	
MTRRphysMask1		If IA32_MTRRCAP[7:0] > 1
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2	
MTRRphysBase2		If IA32_MTRRCAP[7:0] > 2
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2	
MTRRphysMask2		If IA32_MTRRCAP[7:0] > 2
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3	
MTRRphysBase3		If IA32_MTRRCAP[7:0] > 3
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3	
MTRRphysMask3		If IA32_MTRRCAP[7:0] > 3
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4	
MTRRphysBase4		If IA32_MTRRCAP[7:0] > 4
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4	
MTRRphysMask4		If IA32_MTRRCAP[7:0] > 4
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5	
MTRRphysBase5		If IA32_MTRRCAP[7:0] > 5
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5	
MTRRphysMask5		If IA32_MTRRCAP[7:0] > 5
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6	
MTRRphysBase6		If IA32_MTRRCAP[7:0] > 6
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6	
MTRRphysMask6		If IA32_MTRRCAP[7:0] > 6
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7	
MTRRphysBase7		If IA32_MTRRCAP[7:0] > 7
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7	
MTRRphysMask7		If IA32_MTRRCAP[7:0] > 7
Register Address: 210H, 528	IA32_MTRR_PHYSBASE8	
MTRRphysBase8		If IA32_MTRRCAP[7:0] > 8
Register Address: 211H, 529	IA32_MTRR_PHYSMASK8	
MTRRphysMask8		If IA32_MTRRCAP[7:0] > 8
Register Address: 212H, 530	IA32_MTRR_PHYSBASE9	
MTRRphysBase9		If IA32_MTRRCAP[7:0] > 9
Register Address: 213H, 531	IA32_MTRR_PHYSMASK9	
MTRRphysMask9		If IA32_MTRRCAP[7:0] > 9
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000	
MTRRfix64K_00000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
MTRRfix16K_80000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000	
MTRRfix16K_A0000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000)	
See Section 13.11.2.2, "Fixed Range MTRRs."		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000	
MTRRfix4K_C8000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000	
MTRRfix4K_D0000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000	
MTRRfix4K_D8000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000	
MTRRfix4K_E0000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000	
MTRRfix4K_E8000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000	
MTRRfix4K_F0000		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000	
MTRRfix4K_F8000.		If CPUID.01H: EDX.MTRR[12] =1
Register Address: 277H, 631	IA32_PAT	
IA32_PAT (R/W)		If CPUID.01H: EDX.MTRR[16] =1
2:0	PA0	
7:3	Reserved.	
10:8	PA1	
15:11	Reserved.	
18:16	PA2	
23:19	Reserved.	
26:24	PA3	
31:27	Reserved.	
34:32	PA4	
39:35	Reserved.	
42:40	PA5	
47:43	Reserved.	
50:48	PA6	
55:51	Reserved.	
58:56	PA7	
63:59	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 280H, 640	IA32_MC0_CTL2	
MSR to enable/disable CMCI capability for bank 0. (R/W) See Section 17.3.2.5, "IA32_MC**i**_CTL2 MSRs."		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 0
14:0	Corrected error count threshold.	
29:15	Reserved.	
30	CMCI_EN	
63:31	Reserved.	
Register Address: 281H, 641	IA32_MC1_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 1
Register Address: 282H, 642	IA32_MC2_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 2
Register Address: 283H, 643	IA32_MC3_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 3
Register Address: 284H, 644	IA32_MC4_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 4
Register Address: 285H, 645	IA32_MC5_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 5
Register Address: 286H, 646	IA32_MC6_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 6
Register Address: 287H, 647	IA32_MC7_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 7
Register Address: 288H, 648	IA32_MC8_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 8
Register Address: 289H, 649	IA32_MC9_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 9
Register Address: 28AH, 650	IA32_MC10_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 10
Register Address: 28BH, 651	IA32_MC11_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 11
Register Address: 28CH, 652	IA32_MC12_CTL2	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 12
Register Address: 28DH, 653		IA32_MC13_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 13
Register Address: 28EH, 654		IA32_MC14_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 14
Register Address: 28FH, 655		IA32_MC15_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 15
Register Address: 290H, 656		IA32_MC16_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 16
Register Address: 291H, 657		IA32_MC17_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 17
Register Address: 292H, 658		IA32_MC18_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 18
Register Address: 293H, 659		IA32_MC19_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 19
Register Address: 294H, 660		IA32_MC20_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 20
Register Address: 295H, 661		IA32_MC21_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 21
Register Address: 296H, 662		IA32_MC22_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 22
Register Address: 297H, 663		IA32_MC23_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 23
Register Address: 298H, 664		IA32_MC24_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 24
Register Address: 299H, 665		IA32_MC25_CTL2
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 25
Register Address: 29AH, 666		IA32_MC26_CTL2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 26
Register Address: 29BH, 667	IA32_MC27_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 27
Register Address: 29CH, 668	IA32_MC28_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 28
Register Address: 29DH, 669	IA32_MC29_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 29
Register Address: 29EH, 670	IA32_MC30_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 30
Register Address: 29FH, 671	IA32_MC31_CTL2	
Same fields as IA32_MC0_CTL2. (R/W)		If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 31
Register Address: 2DCH, 732	IA32_INTEGRITY_STATUS	
IA32_INTEGRITY_STATUS (R/O) Provides status information for integrity features.		If CPUID(EAX=70H, ECX=1H).EDX[24]=1
0	I_AM_IN_STATIC_LSM 0: Static LSM is not active on this logical processor. 1: Static LSM is active on this logical processor.	
63:1	Reserved.	
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE	
MTRRdefType (R/W)		If CPUID.01H: EDX.MTRR[12] =1
2:0	Default Memory Type	
9:3	Reserved.	
10	Fixed Range MTRR Enable	
11	MTRR Enable	
63:12	Reserved.	
Register Address: 309H, 777	IA32_FIXED_CTR0	
Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any.		If CPUID.0AH:EDX[4:0] >0 || CPUID.0AH:ECX[0] = 1 || CPUID.23H.1H:EBX[0] = 1
Register Address: 30AH, 778	IA32_FIXED_CTR1	
Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core.		If CPUID.0AH:EDX[4:0] >1 || CPUID.0AH:ECX[1] = 1 || CPUID.23H.1H:EBX[1] = 1
Register Address: 30BH, 779	IA32_FIXED_CTR2	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref.		If CPUID.0AH:EDX[4:0] >2 || CPUID.0AH:ECX[2] = 1 || CPUID.23H.1H:EBX[2] = 1
Register Address: 30CH, 780	IA32_FIXED_CTR3	
Fixed-Function Performance Counter 3 (R/W): Top-down Microarchitecture Analysis unhalted number of available slots.		If CPUID.0AH:EDX[4:0] >3 || CPUID.0AH:ECX[3] = 1 || CPUID.23H.1H:EBX[3] = 1
Register Address: 30DH, 781	IA32_FIXED_CTR4	
Fixed-Function Performance Counter 4 (R/W): Top-down bad speculation.		If CPUID.0AH:EDX[4:0] >4 || CPUID.0AH:ECX[4] = 1 || CPUID.23H.1H:EBX[4] = 1
47:0	FIXED_COUNTER Top-down bad speculation counter.	
63:46	Reserved.	
Register Address: 30EH, 782	IA32_FIXED_CTR5	
Fixed-Function Performance Counter 5 (R/W): Top-down Frontend Bound.		If CPUID.0AH:EDX[4:0] >5 || CPUID.0AH:ECX[5] = 1 || CPUID.23H.1H:EBX[5] = 1
47:0	FIXED_COUNTER Top-down Frontend Bound counter.	
63:46	Reserved.	
Register Address: 30FH, 783	IA32_FIXED_CTR6	
Fixed-Function Performance Counter 6 (R/W): Top-down retiring.		If CPUID.0AH:EDX[4:0] >6 || CPUID.0AH:ECX[6] = 1 || CPUID.23H.1H:EBX[6] = 1
47:0	FIXED_COUNTER Top-down Retiring counter.	
63:46	Reserved.	
Register Address: 345H, 837	IA32_PERF_CAPABILITIES	
Read Only MSR that enumerates the existence of performance monitoring features. (R/O)		If CPUID.01H: ECX[15] = 1
5:0	LBR format	
6	PEBS Trap	
7	PEBSSaveArchRegs	
11:8	PEBS Record Format	
12	1: Freeze while SMM is supported.	
13	1: Full width of counter writable via IA32_A_PMCx.	
14	PEBS_BASELINE	
15	1: Performance metrics available.	
16	1: PEBS output will be written into the Intel PT trace stream.	If CPUID.0x7.0.EBX[25]=1
17	1: Indicates support for PEBS Retire Latency output.	
18	TSX_ADDRESS	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
19	RDPMC_METRICS_CLEAR	
63:20	Reserved.	
Register Address: 38DH, 909	IA32_FIXED_CTR_CTRL	
Fixed-Function Performance Counter Control (R/W) Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.		If CPUID.0AH: EAX[7:0] > 1
0	EN0_OS: Enable Fixed Counter 0 to count while CPL = 0.	
1	EN0_Usr: Enable Fixed Counter 0 to count while CPL > 0.	
2	AnyThr0: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH:EAX[7:0] > 2 && CPUID.0AH:EDX[15]=0
3	EN0_PMI: Enable PMI when fixed counter 0 overflows.	
4	EN1_OS: Enable Fixed Counter 1to count while CPL = 0.	
5	EN1_Usr: Enable Fixed Counter 1to count while CPL > 0.	
6	AnyThr1: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH:EAX[7:0] > 2 && CPUID.0AH:EDX[15]=0
7	EN1_PMI: Enable PMI when fixed counter 1 overflows.	
8	EN2_OS: Enable Fixed Counter 2 to count while CPL = 0.	
9	EN2_Usr: Enable Fixed Counter 2 to count while CPL > 0.	
10	AnyThr2: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH:EAX[7:0] > 2 && CPUID.0AH:EDX[15]=0
11	EN2_PMI: Enable PMI when fixed counter 2 overflows.	
12	EN3_OS: Enable Fixed Counter 3 to count while CPL = 0.	
13	EN3_Usr: Enable Fixed Counter 3 to count while CPL > 0.	
14	Reserved.	
15	EN3_PMI: Enable PMI when fixed counter 3 overflows.	
63:16	Reserved.	
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
Global Performance Counter Status (R/O)		If CPUID.0AH: EAX[7:0] > 0 ll (CPUID.(EAX=07H, ECX=0):EBX[25] = 1 && CPUID.(EAX=014H, ECX=0):ECX[0] = 1)
0	Ovf_PMC0: Overflow status of IA32_PMC0.	If CPUID.0AH: EAX[15:8] > 0
1	Ovf_PMC1: Overflow status of IA32_PMC1.	If CPUID.0AH: EAX[15:8] > 1
2	Ovf_PMC2: Overflow status of IA32_PMC2.	If CPUID.0AH: EAX[15:8] > 2
3	Ovf_PMC3: Overflow status of IA32_PMC3.	If CPUID.0AH: EAX[15:8] > 3

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
n	Ovf_PMCn: Overflow status of IA32_PMCn.	If CPUID.0AH: EAX[15:8] > n
31:n+1	Reserved.	
32	Ovf_FixedCtr0: Overflow status of IA32_FIXED_CTR0.	If CPUID.0AH: EAX[7:0] > 1
33	Ovf_FixedCtr1: Overflow status of IA32_FIXED_CTR1.	If CPUID.0AH: EAX[7:0] > 1
34	Ovf_FixedCtr2: Overflow status of IA32_FIXED_CTR2.	If CPUID.0AH: EAX[7:0] > 1
32+m	Ovf_FixedCtrm: Overflow status of IA32_FIXED_CTRm.	If CPUID.0AH:ECX[m] == 1 || CPUID.0AH:EDX[4:0] > m
47:33+m	Reserved.	
48	OVF_PERF_METRICS: If this bit is set, it indicates that PERF_METRIC counter has overflowed and a PMI is triggered; however, an overflow of fixed counter 3 should normally happen first. If this bit is clear no overflow occurred.	
54:49	Reserved.	
55	Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer that was completely filled.	If CPUID.(EAX=07H, ECX=0):EBX[25] = 1 && CPUID.(EAX=014H, ECX=0):ECX[0] = 1
57:56	Reserved.	
58	LBR_Frz. LBRs are frozen due to: ▪ IA32_DEBUGCTL.FREEZE_LBR_ON_PMI=1. ▪ The LBR stack overflowed.	If CPUID.0AH: EAX[7:0] > 3
59	CTR_Frz. Performance counters in the core PMU are frozen due to: ▪ IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI=1. ▪ One or more core PMU counters overflowed.	If CPUID.0AH: EAX[7:0] > 3
60	ASCI: Data in the performance counters in the core PMU may include contributions from the direct or indirect operation Intel SGX to protect an enclave.	If the processor supports Intel® SGX.
61	Ovf_Uncore: Uncore counter overflow status.	If CPUID.0AH: EAX[7:0] > 2
62	OvfBuf: DS SAVE area Buffer overflow status.	If CPUID.0AH: EAX[7:0] > 0
63	CondChgd: Status bits of this register have changed.	If CPUID.0AH: EAX[7:0] > 0
Register Address: 38FH, 911		IA32_PERF_GLOBAL_CTRL
Global Performance Counter Control (R/W) Counter increments while the result of ANDing the respective enable bit in this MSR with the corresponding OS or USR bits in the general-purpose or fixed counter control MSR is true.		If CPUID.0AH: EAX[7:0] > 0
0	EN_PMC0	If CPUID.0AH: EAX[15:8] > 0
1	EN_PMC1	If CPUID.0AH: EAX[15:8] > 1
2	EN_PMC2	If CPUID.0AH: EAX[15:8] > 2
n	EN_PMCn	If CPUID.0AH: EAX[15:8] > n
31:n+1	Reserved.	
32	EN_FIXED_CTR0	If CPUID.0AH: EDX[4:0] > 0
33	EN_FIXED_CTR1	If CPUID.0AH: EDX[4:0] > 1
34	EN_FIXED_CTR2	If CPUID.0AH: EDX[4:0] > 2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
32+m	EN_FIXED_CTRm	If CPUID.0AH:ECX[m] == 1 || CPUID.0AH:EDX[4:0] > m
47:33+m	Reserved.	
48	EN_PERF_METRICS: If this bit is set and fixed counter 3 is effectively enabled, built-in performance metrics are enabled.	
63:49	Reserved.	
Register Address: 390H, 912	IA32_PERF_GLOBAL_STATUS_RESET	
Global Performance Counter Overflow Reset Control (R/W)		If CPUID.0AH: EAX[7:0] > 3 II (CPUID.(EAX=07H, ECX=0):EBX[25] = 1 && CPUID.(EAX=14H, ECX=0):ECX[0] = 1)
0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1
2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
31:n	Reserved.	
32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
32+m	Set 1 to Clear Ovf_FIXED_CTRm bit.	If CPUID.0AH:ECX[m] == 1 || CPUID.0AH:EDX[4:0] > m
47:33+m	Reserved.	
48	RESET_OVF_PERF_METRICS: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.	
54:49	Reserved.	
55	Set 1 to Clear Trace_ToPA_PMI bit.	If CPUID.(EAX=07H, ECX=0):EBX[25] = 1 && CPUID.(EAX=014H, ECX=0):ECX[0] = 1
57:56	Reserved.	
58	Set 1 to Clear LBR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
59	Set 1 to Clear CTR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
60	Set 1 to Clear ASCI bit.	If the processor supports Intel® SGX.
61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
62	Set 1 to Clear OvfBuf bit.	If CPUID.0AH: EAX[7:0] > 0
63	Set 1 to clear CondChgd bit.	If CPUID.0AH: EAX[7:0] > 0
Register Address: 391H, 913	IA32_PERF_GLOBAL_STATUS_SET	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Global Performance Counter Overflow Set Control (R/W)		If CPUID.0AH: EAX[7:0] > 3 II (CPUID.(EAX=07H, ECX=0):EBX[25] = 1 && CPUID.(EAX=014H, ECX=0):ECX[0] = 1)
0	Set 1 to cause Ovf_PMC0 = 1.	If CPUID.0AH: EAX[7:0] > 3
1	Set 1 to cause Ovf_PMC1 = 1.	If CPUID.0AH: EAX[15:8] > 1
2	Set 1 to cause Ovf_PMC2 = 1.	If CPUID.0AH: EAX[15:8] > 2
n	Set 1 to cause Ovf_PMCn = 1.	If CPUID.0AH: EAX[15:8] > n
31:n	Reserved.	
32	Set 1 to cause Ovf_FIXED_CTR0 = 1.	If CPUID.0AH: EAX[7:0] > 3
33	Set 1 to cause Ovf_FIXED_CTR1 = 1.	If CPUID.0AH: EAX[7:0] > 3
34	Set 1 to cause Ovf_FIXED_CTR2 = 1.	If CPUID.0AH: EAX[7:0] > 3
32+m	Set 1 to cause Ovf_FIXED_CTRm = 1.	If CPUID.0AH:ECX[m] == 1 || CPUID.0AH:EDX[4:0] > m
47:33+m	Reserved.	
48	SET_OVF_PERF_METRICS: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.	
54:49	Reserved.	
55	Set 1 to cause Trace_ToPA_PMI = 1.	If CPUID.(EAX=07H, ECX=0):EBX[25] = 1 && CPUID.(EAX=014H, ECX=0):ECX[0] = 1
57:56	Reserved.	
58	Set 1 to cause LBR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
59	Set 1 to cause CTR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
60	Set 1 to cause ASCI = 1.	If the processor supports Intel® SGX.
61	Set 1 to cause Ovf_Uncore = 1.	If CPUID.0AH: EAX[7:0] > 3
62	Set 1 to cause OvfBuf = 1.	If CPUID.0AH: EAX[7:0] > 3
63	Reserved.	
Register Address: 392H, 914		IA32_PERF_GLOBAL_INUSE
Indicator that core PerfMon interface is in use. (R/O)		If CPUID.0AH: EAX[7:0] > 3
0	IA32_PERFEVTSEL0 in use.	
1	IA32_PERFEVTSEL1 in use.	If CPUID.0AH: EAX[15:8] > 1
2	IA32_PERFEVTSEL2 in use.	If CPUID.0AH: EAX[15:8] > 2
n	IA32_PERFEVTSELn in use.	If CPUID.0AH: EAX[15:8] > n
31:n+1	Reserved.	
32	IA32_FIXED_CTR0 in use.	
33	IA32_FIXED_CTR1 in use.	
34	IA32_FIXED_CTR2 in use.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
32+m	IA32_FIXED_CTRm in use.	
62:33+m	Reserved or model specific.	
63	PMI in use.	
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE	
PEBS Control (R/W)		
0	Enable PEBS on IA32_PMC0.	06_0FH
3:1	Reserved or model specific.	
31:4	Reserved.	
35:32	Reserved or model specific.	
63:36	Reserved.	
Register Address: 400H, 1024	IA32_MC0_CTL	
MC0_CTL		If IA32_MCG_CAP.CNT >0
Register Address: 401H, 1025	IA32_MC0_STATUS	
MC0_STATUS		If IA32_MCG_CAP.CNT >0
Register Address: 402H, 1026	IA32_MC0_ADDR[1]	
MC0_ADDR		If IA32_MCG_CAP.CNT >0
Register Address: 403H, 1027	IA32_MC0_MISC	
MC0_MISC		If IA32_MCG_CAP.CNT >0
Register Address: 404H, 1028	IA32_MC1_CTL	
MC1_CTL		If IA32_MCG_CAP.CNT >1
Register Address: 405H, 1029	IA32_MC1_STATUS	
MC1_STATUS		If IA32_MCG_CAP.CNT >1
Register Address: 406H, 1030	IA32_MC1_ADDR[2]	
MC1_ADDR		If IA32_MCG_CAP.CNT >1
Register Address: 407H, 1031	IA32_MC1_MISC	
MC1_MISC		If IA32_MCG_CAP.CNT >1
Register Address: 408H, 1032	IA32_MC2_CTL	
MC2_CTL		If IA32_MCG_CAP.CNT >2
Register Address: 409H, 1033	IA32_MC2_STATUS	
MC2_STATUS		If IA32_MCG_CAP.CNT >2
Register Address: 40AH, 1034	IA32_MC2_ADDR[1]	
MC2_ADDR		If IA32_MCG_CAP.CNT >2
Register Address: 40BH, 1035	IA32_MC2_MISC	
MC2_MISC		If IA32_MCG_CAP.CNT >2
Register Address: 40CH, 1036	IA32_MC3_CTL	
MC3_CTL		If IA32_MCG_CAP.CNT >3
Register Address: 40DH, 1037	IA32_MC3_STATUS	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
MC3_STATUS		If IA32_MCG_CAP.CNT >3
Register Address: 40EH, 1038	IA32_MC3_ADDR[1]	
MC3_ADDR		If IA32_MCG_CAP.CNT >3
Register Address: 40FH, 1039	IA32_MC3_MISC	
MC3_MISC		If IA32_MCG_CAP.CNT >3
Register Address: 410H, 1040	IA32_MC4_CTL	
MC4_CTL		If IA32_MCG_CAP.CNT >4
Register Address: 411H, 1041	IA32_MC4_STATUS	
MC4_STATUS		If IA32_MCG_CAP.CNT >4
Register Address: 412H, 1042	IA32_MC4_ADDR[1]	
MC4_ADDR		If IA32_MCG_CAP.CNT >4
Register Address: 413H, 1043	IA32_MC4_MISC	
MC4_MISC		If IA32_MCG_CAP.CNT >4
Register Address: 414H, 1044	IA32_MC5_CTL	
MC5_CTL		If IA32_MCG_CAP.CNT >5
Register Address: 415H, 1045	IA32_MC5_STATUS	
MC5_STATUS		If IA32_MCG_CAP.CNT >5
Register Address: 416H, 1046	IA32_MC5_ADDR[1]	
MC5_ADDR		If IA32_MCG_CAP.CNT >5
Register Address: 417H, 1047	IA32_MC5_MISC	
MC5_MISC		If IA32_MCG_CAP.CNT >5
Register Address: 418H, 1048	IA32_MC6_CTL	
MC6_CTL		If IA32_MCG_CAP.CNT >6
Register Address: 419H, 1049	IA32_MC6_STATUS	
MC6_STATUS		If IA32_MCG_CAP.CNT >6
Register Address: 41AH, 1050	IA32_MC6_ADDR[1]	
MC6_ADDR		If IA32_MCG_CAP.CNT >6
Register Address: 41BH, 1051	IA32_MC6_MISC	
MC6_MISC		If IA32_MCG_CAP.CNT >6
Register Address: 41CH, 1052	IA32_MC7_CTL	
MC7_CTL		If IA32_MCG_CAP.CNT >7
Register Address: 41DH, 1053	IA32_MC7_STATUS	
MC7_STATUS		If IA32_MCG_CAP.CNT >7
Register Address: 41EH, 1054	IA32_MC7_ADDR[1]	
MC7_ADDR		If IA32_MCG_CAP.CNT >7
Register Address: 41FH, 1055	IA32_MC7_MISC	
MC7_MISC		If IA32_MCG_CAP.CNT >7

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 420H, 1056	IA32_MC8_CTL	
MC8_CTL		If IA32_MCG_CAP.CNT >8
Register Address: 421H, 1057	IA32_MC8_STATUS	
MC8_STATUS		If IA32_MCG_CAP.CNT >8
Register Address: 422H, 1058	IA32_MC8_ADDR[1]	
MC8_ADDR		If IA32_MCG_CAP.CNT >8
Register Address: 423H, 1059	IA32_MC8_MISC	
MC8_MISC		If IA32_MCG_CAP.CNT >8
Register Address: 424H, 1060	IA32_MC9_CTL	
MC9_CTL		If IA32_MCG_CAP.CNT >9
Register Address: 425H, 1061	IA32_MC9_STATUS	
MC9_STATUS		If IA32_MCG_CAP.CNT >9
Register Address: 426H, 1062	IA32_MC9_ADDR[1]	
MC9_ADDR		If IA32_MCG_CAP.CNT >9
Register Address: 427H, 1063	IA32_MC9_MISC	
MC9_MISC		If IA32_MCG_CAP.CNT >9
Register Address: 428H, 1064	IA32_MC10_CTL	
MC10_CTL		If IA32_MCG_CAP.CNT >10
Register Address: 429H, 1065	IA32_MC10_STATUS	
MC10_STATUS		If IA32_MCG_CAP.CNT >10
Register Address: 42AH, 1066	IA32_MC10_ADDR[1]	
MC10_ADDR		If IA32_MCG_CAP.CNT >10
Register Address: 42BH, 1067	IA32_MC10_MISC	
MC10_MISC		If IA32_MCG_CAP.CNT >10
Register Address: 42CH, 1068	IA32_MC11_CTL	
MC11_CTL		If IA32_MCG_CAP.CNT >11
Register Address: 42DH, 1069	IA32_MC11_STATUS	
MC11_STATUS		If IA32_MCG_CAP.CNT >11
Register Address: 42EH, 1070	IA32_MC11_ADDR[1]	
MC11_ADDR		If IA32_MCG_CAP.CNT >11
Register Address: 42FH, 1071	IA32_MC11_MISC	
MC11_MISC		If IA32_MCG_CAP.CNT >11
Register Address: 430H, 1072	IA32_MC12_CTL	
MC12_CTL		If IA32_MCG_CAP.CNT >12
Register Address: 431H, 1073	IA32_MC12_STATUS	
MC12_STATUS		If IA32_MCG_CAP.CNT >12
Register Address: 432H, 1074	IA32_MC12_ADDR[1]	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
MC12_ADDR		If IA32_MCG_CAP.CNT >12
Register Address: 433H, 1075	IA32_MC12_MISC	
MC12_MISC		If IA32_MCG_CAP.CNT >12
Register Address: 434H, 1076	IA32_MC13_CTL	
MC13_CTL		If IA32_MCG_CAP.CNT >13
Register Address: 435H, 1077	IA32_MC13_STATUS	
MC13_STATUS		If IA32_MCG_CAP.CNT >13
Register Address: 436H, 1078	IA32_MC13_ADDR[1]	
MC13_ADDR		If IA32_MCG_CAP.CNT >13
Register Address: 437H, 1079	IA32_MC13_MISC	
MC13_MISC		If IA32_MCG_CAP.CNT >13
Register Address: 438H, 1080	IA32_MC14_CTL	
MC14_CTL		If IA32_MCG_CAP.CNT >14
Register Address: 439H, 1081	IA32_MC14_STATUS	
MC14_STATUS		If IA32_MCG_CAP.CNT >14
Register Address: 43AH, 1082	IA32_MC14_ADDR[1]	
MC14_ADDR		If IA32_MCG_CAP.CNT >14
Register Address: 43BH, 1083	IA32_MC14_MISC	
MC14_MISC		If IA32_MCG_CAP.CNT >14
Register Address: 43CH, 1084	IA32_MC15_CTL	
MC15_CTL		If IA32_MCG_CAP.CNT >15
Register Address: 43DH, 1085	IA32_MC15_STATUS	
MC15_STATUS		If IA32_MCG_CAP.CNT >15
Register Address: 43EH, 1086	IA32_MC15_ADDR[1]	
MC15_ADDR		If IA32_MCG_CAP.CNT >15
Register Address: 43FH, 1087	IA32_MC15_MISC	
MC15_MISC		If IA32_MCG_CAP.CNT >15
Register Address: 440H, 1088	IA32_MC16_CTL	
MC16_CTL		If IA32_MCG_CAP.CNT >16
Register Address: 441H, 1089	IA32_MC16_STATUS	
MC16_STATUS		If IA32_MCG_CAP.CNT >16
Register Address: 442H, 1090	IA32_MC16_ADDR[1]	
MC16_ADDR		If IA32_MCG_CAP.CNT >16
Register Address: 443H, 1091	IA32_MC16_MISC	
MC16_MISC		If IA32_MCG_CAP.CNT >16
Register Address: 444H, 1092	IA32_MC17_CTL	
MC17_CTL		If IA32_MCG_CAP.CNT >17

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 445H, 1093	IA32_MC17_STATUS	
MC17_STATUS		If IA32_MCG_CAP.CNT >17
Register Address: 446H, 1094	IA32_MC17_ADDR[1]	
MC17_ADDR		If IA32_MCG_CAP.CNT >17
Register Address: 447H, 1095	IA32_MC17_MISC	
MC17_MISC		If IA32_MCG_CAP.CNT >17
Register Address: 448H, 1096	IA32_MC18_CTL	
MC18_CTL		If IA32_MCG_CAP.CNT >18
Register Address: 449H, 1097	IA32_MC18_STATUS	
MC18_STATUS		If IA32_MCG_CAP.CNT >18
Register Address: 44AH, 1098	IA32_MC18_ADDR[1]	
MC18_ADDR		If IA32_MCG_CAP.CNT >18
Register Address: 44BH, 1099	IA32_MC18_MISC	
MC18_MISC		If IA32_MCG_CAP.CNT >18
Register Address: 44CH, 1100	IA32_MC19_CTL	
MC19_CTL		If IA32_MCG_CAP.CNT >19
Register Address: 44DH, 1101	IA32_MC19_STATUS	
MC19_STATUS		If IA32_MCG_CAP.CNT >19
Register Address: 44EH, 1102	IA32_MC19_ADDR[1]	
MC19_ADDR		If IA32_MCG_CAP.CNT >19
Register Address: 44FH, 1103	IA32_MC19_MISC	
MC19_MISC		If IA32_MCG_CAP.CNT >19
Register Address: 450H, 1104	IA32_MC20_CTL	
MC20_CTL		If IA32_MCG_CAP.CNT >20
Register Address: 451H, 1105	IA32_MC20_STATUS	
MC20_STATUS		If IA32_MCG_CAP.CNT >20
Register Address: 452H, 1106	IA32_MC20_ADDR[1]	
MC20_ADDR		If IA32_MCG_CAP.CNT >20
Register Address: 453H, 1107	IA32_MC20_MISC	
MC20_MISC		If IA32_MCG_CAP.CNT >20
Register Address: 454H, 1108	IA32_MC21_CTL	
MC21_CTL		If IA32_MCG_CAP.CNT >21
Register Address: 455H, 1109	IA32_MC21_STATUS	
MC21_STATUS		If IA32_MCG_CAP.CNT >21
Register Address: 456H, 1110	IA32_MC21_ADDR[1]	
MC21_ADDR		If IA32_MCG_CAP.CNT >21
Register Address: 457H, 1111	IA32_MC21_MISC	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
MC21_MISC		If IA32_MCG_CAP.CNT >21
Register Address: 458H, 1112	IA32_MC22_CTL	
MC22_CTL		If IA32_MCG_CAP.CNT >22
Register Address: 459H, 1113	IA32_MC22_STATUS	
MC22_STATUS		If IA32_MCG_CAP.CNT >22
Register Address: 45AH, 1114	IA32_MC22_ADDR[1]	
MC22_ADDR		If IA32_MCG_CAP.CNT >22
Register Address: 45BH, 1115	IA32_MC22_MISC	
MC22_MISC		If IA32_MCG_CAP.CNT >22
Register Address: 45CH, 1116	IA32_MC23_CTL	
MC23_CTL		If IA32_MCG_CAP.CNT >23
Register Address: 45DH, 1117	IA32_MC23_STATUS	
MC23_STATUS		If IA32_MCG_CAP.CNT >23
Register Address: 45EH, 1118	IA32_MC23_ADDR[1]	
MC23_ADDR		If IA32_MCG_CAP.CNT >23
Register Address: 45FH, 1119	IA32_MC23_MISC	
MC23_MISC		If IA32_MCG_CAP.CNT >23
Register Address: 460H, 1120	IA32_MC24_CTL	
MC24_CTL		If IA32_MCG_CAP.CNT >24
Register Address: 461H, 1121	IA32_MC24_STATUS	
MC24_STATUS		If IA32_MCG_CAP.CNT >24
Register Address: 462H, 1122	IA32_MC24_ADDR[1]	
MC24_ADDR		If IA32_MCG_CAP.CNT >24
Register Address: 463H, 1123	IA32_MC24_MISC	
MC24_MISC		If IA32_MCG_CAP.CNT >24
Register Address: 464H, 1124	IA32_MC25_CTL	
MC25_CTL		If IA32_MCG_CAP.CNT >25
Register Address: 465H, 1125	IA32_MC25_STATUS	
MC25_STATUS		If IA32_MCG_CAP.CNT >25
Register Address: 466H, 1126	IA32_MC25_ADDR[1]	
MC25_ADDR		If IA32_MCG_CAP.CNT >25
Register Address: 467H, 1127	IA32_MC25_MISC	
MC25_MISC		If IA32_MCG_CAP.CNT >25
Register Address: 468H, 1128	IA32_MC26_CTL	
MC26_CTL		If IA32_MCG_CAP.CNT >26
Register Address: 469H, 1129	IA32_MC26_STATUS	
MC26_STATUS		If IA32_MCG_CAP.CNT >26

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 46AH, 1130	IA32_MC26_ADDR[1]	
MC26_ADDR		If IA32_MCG_CAP.CNT >26
Register Address: 46BH, 1131	IA32_MC26_MISC	
MC26_MISC		If IA32_MCG_CAP.CNT >26
Register Address: 46CH, 1132	IA32_MC27_CTL	
MC27_CTL		If IA32_MCG_CAP.CNT >27
Register Address: 46DH, 1133	IA32_MC27_STATUS	
MC27_STATUS		If IA32_MCG_CAP.CNT >27
Register Address: 46EH, 1134	IA32_MC27_ADDR[1]	
MC27_ADDR		If IA32_MCG_CAP.CNT >27
Register Address: 46FH, 1135	IA32_MC27_MISC	
MC27_MISC		If IA32_MCG_CAP.CNT >27
Register Address: 470H, 1136	IA32_MC28_CTL	
MC28_CTL		If IA32_MCG_CAP.CNT >28
Register Address: 471H, 1137	IA32_MC28_STATUS	
MC28_STATUS		If IA32_MCG_CAP.CNT >28
Register Address: 472H, 1138	IA32_MC28_ADDR[1]	
MC28_ADDR		If IA32_MCG_CAP.CNT >28
Register Address: 473H, 1139	IA32_MC28_MISC	
MC28_MISC		If IA32_MCG_CAP.CNT >28
Register Address: 474H, 1140	IA32_MC29_CTL	
MC29_CTL		If IA32_MCG_CAP.CNT >29
Register Address: 475H, 1141	IA32_MC29_STATUS	
MC29_STATUS		If IA32_MCG_CAP.CNT >29
Register Address: 476H, 1142	IA32_MC29_ADDR	
MC29_ADDR		If IA32_MCG_CAP.CNT >29
Register Address: 477H, 1143	IA32_MC29_MISC	
MC29_MISC		If IA32_MCG_CAP.CNT >29
Register Address: 478H, 1144	IA32_MC30_CTL	
MC30_CTL		If IA32_MCG_CAP.CNT >30
Register Address: 479H, 1145	IA32_MC30_STATUS	
MC30_STATUS		If IA32_MCG_CAP.CNT >30
Register Address: 47AH, 1146	IA32_MC30_ADDR	
MC30_ADDR		If IA32_MCG_CAP.CNT >30
Register Address: 47BH, 1147	IA32_MC30_MISC	
MC30_MISC		If IA32_MCG_CAP.CNT >30
Register Address: 47CH, 1148	IA32_MC31_CTL	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
MC31_CTL		If IA32_MCG_CAP.CNT >31
Register Address: 47DH, 1149	IA32_MC31_STATUS	
MC31_STATUS		If IA32_MCG_CAP.CNT >31
Register Address: 47EH, 1150	IA32_MC31_ADDR	
MC31_ADDR		If IA32_MCG_CAP.CNT >31
Register Address: 47FH, 1151	IA32_MC31_MISC	
MC31_MISC		If IA32_MCG_CAP.CNT >31
Register Address: 480H, 1152	IA32_VMX_BASIC	
Reporting Register of Basic VMX Capabilities (R/O) See Appendix A.1, “Basic VMX Information.”		If CPUID.01H:ECX.[5] = 1
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Appendix A.3.1, “Pin-Based VM-Execution Controls.”		If CPUID.01H:ECX.[5] = 1
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3.2, “Primary Processor-Based VM-Execution Controls.”		If CPUID.01H:ECX.[5] = 1
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	
Capability Reporting Register of Primary VM-Exit Controls (R/O) See Appendix A.4.1, “Primary VM-Exit Controls.”		If CPUID.01H:ECX.[5] = 1
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Controls (R/O) See Appendix A.5, “VM-Entry Controls.”		If CPUID.01H:ECX.[5] = 1
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Appendix A.6, “Miscellaneous Data.”		If CPUID.01H:ECX.[5] = 1
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Appendix A.7, “VMX-Fixed Bits in CR0.”		If CPUID.01H:ECX.[5] = 1
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Appendix A.7, “VMX-Fixed Bits in CR0.”		If CPUID.01H:ECX.[5] = 1
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Appendix A.8, “VMX-Fixed Bits in CR4.”		If CPUID.01H:ECX.[5] = 1
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Appendix A.8, “VMX-Fixed Bits in CR4.”		If CPUID.01H:ECX.[5] = 1
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Capability Reporting Register of VMCS Field Enumeration (R/O) See Appendix A.9, “VMCS Enumeration.”		If CPUID.01H:ECX.[5] = 1
Register Address: 48BH, 1163		IA32_VMX_PROCBASED_CTLS2
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3.3, “Secondary Processor-Based VM-Execution Controls.”		If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63])
Register Address: 48CH, 1164		IA32_VMX_EPT_VPID_CAP
Capability Reporting Register of EPT and VPID (R/O) See Appendix A.10, “VPID and EPT Capabilities.”		If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63] && (IA32_VMX_PROCBASED_CTLS2[3 3] || IA32_VMX_PROCBASED_CTLS2[3 7]))
Register Address: 48DH, 1165		IA32_VMX_TRUE_PINBASED_CTLS
Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Appendix A.3.1, “Pin-Based VM-Execution Controls.”		If (CPUID.01H:ECX.[5] && IA32_VMX_BASIC[55])
Register Address: 48EH, 1166		IA32_VMX_TRUE_PROCBASED_CTLS
Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Appendix A.3.2, “Primary Processor-Based VM-Execution Controls.”		If(CPUID.01H:ECX.[5] && IA32_VMX_BASIC[55])
Register Address: 48FH, 1167		IA32_VMX_TRUE_EXIT_CTLS
Capability Reporting Register of VM-Exit Flex Controls (R/O) See Appendix A.4, “VM-Exit Controls.”		If(CPUID.01H:ECX.[5] && IA32_VMX_BASIC[55])
Register Address: 490H, 1168		IA32_VMX_TRUE_ENTRY_CTLS
Capability Reporting Register of VM-Entry Flex Controls (R/O) See Appendix A.5, “VM-Entry Controls.”		If(CPUID.01H:ECX.[5] && IA32_VMX_BASIC[55])
Register Address: 491H, 1169		IA32_VMX_VMFUNC
Capability Reporting Register of VM-Function Controls (R/O)		If(CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63] && IA32_VMX_PROCBASED_CTLS2[4 5])
Register Address: 492H, 1170		IA32_VMX_PROCBASED_CTLS3
Capability Reporting Register of Tertiary Processor-Based VM-Execution Controls (R/O) See Appendix A.3.4, “Tertiary Processor-Based VM-Execution Controls.”		If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[49])
Register Address: 493H, 1171		IA32_VMX_EXIT_CTLS2
Capability Reporting Register of Secondary VM-Exit Controls (R/O) See Appendix A.4.2, “Secondary VM-Exit Controls.”		If (CPUID.01H:ECX.[5] && IA32_VMX_EXIT_CTLS[63])
Register Address: 4C1H, 1217		IA32_A_PMC0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Full Width Writable IA32_PMC0 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 0) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[0] = 1
Register Address: 4C2H, 1218		IA32_A_PMC1
Full Width Writable IA32_PMC1 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 1) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[1] = 1
Register Address: 4C3H, 1219		IA32_A_PMC2
Full Width Writable IA32_PMC2 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 2) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[2] = 1
Register Address: 4C4H, 1220		IA32_A_PMC3
Full Width Writable IA32_PMC3 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 3) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[3] = 1
Register Address: 4C5H, 1221		IA32_A_PMC4
Full Width Writable IA32_PMC4 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 4) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[4] = 1
Register Address: 4C6H, 1222		IA32_A_PMC5
Full Width Writable IA32_PMC5 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 5) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[5] = 1
Register Address: 4C7H, 1223		IA32_A_PMC6
Full Width Writable IA32_PMC6 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 6) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[6] = 1
Register Address: 4C8H, 1224		IA32_A_PMC7
Full Width Writable IA32_PMC7 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 7) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[7] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 4C9H, 1225	IA32_A_PMC8	
Full Width Writable IA32_PMC8 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 8) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[8] = 1
Register Address: 4CAH, 1226	IA32_A_PMC9	
Full Width Writable IA32_PMC9 Alias (R/W)		If (CPUID.0AH:EAX[15:8] > 9) && IA32_PERF_CAPABILITIES[13] = 1) || CPUID.(EAX=23H,ECX=1):EAX[9] = 1
Register Address: 4D0H, 1232	IA32_MCG_EXT_CTL	
Allows software to signal some MCEs to only a single logical processor in the system. (R/W) See Section 17.3.1.4, "IA32_MCG_EXT_CTL MSR."		If IA32_MCG_CAP.LMCE_P =1
0	LMCE_EN Enable / Disable local machine check exception.	
63:1	Reserved.	
Register Address: 500H, 1280	IA32_SGX_SVN_STATUS	
Status and SVN Threshold of SGX Support for ACM (R/O)		If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
0	Lock.	See Section 40.11.3, "Interactions with Authenticated Code Modules (ACMs)."
15:1	Reserved.	
23:16	SGX_SVN_SINIT	See Section 40.11.3, "Interactions with Authenticated Code Modules (ACMs)."
63:24	Reserved.	
Register Address: 560H, 1376	IA32_RTIT_OUTPUT_BASE	
Trace Output Base Register (R/W)		If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0):ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0):ECX[2] = 1)))
6:0	Reserved.	
MAXPHYADDR[4]-1:7	Base physical address.	
63:MAXPHYADDR	Reserved.	
Register Address: 561H, 1377	IA32_RTIT_OUTPUT_MASK_PTRS	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Trace Output Mask Pointers Register (R/W)		If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0):ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0):ECX[2] = 1)))
6:0	Reserved.	
31:7	MaskOrTableOffset.	
63:32	Output Offset.	
Register Address: 570H, 1392		IA32_RTIT_CTL
Trace Control Register (R/W)		If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
0	TraceEn	
1	CYCEn	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
2	OS	
3	User	
4	PwrEvtEn	If (CPUID.(EAX=07H, ECX=1):EBX[5] = 1)
5	FUPonPTW	If (CPUID.(EAX=07H, ECX=1):EBX[4] = 1)
6	FabricEn	If (CPUID.(EAX=07H, ECX=0):ECX[3] = 1)
7	CR3Filter	If (CPUID.(EAX=14H, ECX=0):EBX[0] = 1)
8	ToPA	
9	MTCEn	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
10	TSCEn	
11	DisRETC	
12	PTWEn	If (CPUID.(EAX=07H, ECX=1):EBX[4] = 1)
13	BranchEn	
17:14	MTCFreq.	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
18	Reserved, must be zero.	
22:19	CycThresh	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
23	Reserved, must be zero.	
27:24	PSBFreq	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
30:28	Reserved, must be zero.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
31	EventEn	If (CPUID.(EAX=14H, ECX=0):EBX[7] = 1)
35:32	ADDR0_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
39:36	ADDR1_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
43:40	ADDR2_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
47:44	ADDR3_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
54:48	Reserved, must be zero.	
55	DisTNT	If (CPUID.(EAX=14H, ECX=0):EBX[8] = 1)
56	InjectPsbPmiOnEnable	If (CPUID.(EAX=07H, ECX=1):EBX[6] = 1)
63:57	Reserved, must be zero.	
Register Address: 571H, 1393	IA32_RTIT_STATUS	
Tracing Status Register (R/W)		If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
0	FilterEn (writes ignored).	If (CPUID.(EAX=07H, ECX=0):EBX[2] = 1)
1	ContexEn (writes ignored).	
2	TriggerEn (writes ignored).	
3	Reserved.	
4	Error	
5	Stopped	
6	PendPSB	If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)
7	PendToPAPMI	If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)
31:8	Reserved, must be zero.	
48:32	PacketByteCnt	If (CPUID.(EAX=07H, ECX=0):EBX[1] > 3)
63:49	Reserved.	
Register Address: 572H, 1394	IA32_RTIT_CR3_MATCH	
Trace Filter CR3 Match Register (R/W)		If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
4:0	Reserved.	
63:5	CR3[63:5] value to match.	
Register Address: 580H, 1408	IA32_RTIT_ADDR0_A	
Region 0 Start Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 581H, 1409	IA32_RTIT_ADDR0_B	
Region 0 End Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 582H, 1410	IA32_RTIT_ADDR1_A	
Region 1 Start Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 583H, 1411	IA32_RTIT_ADDR1_B	
Region 1 End Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 584H, 1412	IA32_RTIT_ADDR2_A	
Region 2 Start Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 585H, 1413	IA32_RTIT_ADDR2_B	
Region 2 End Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 586H, 1414	IA32_RTIT_ADDR3_A	
Region 3 Start Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 587H, 1415	IA32_RTIT_ADDR3_B	
Region 3 End Address (R/W)		If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
47:0	Virtual Address.	
63:48	SignExt_VA	
Register Address: 600H, 1536	IA32_DS_AREA	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
DS Save Area (R/W) Points to the linear address of the first byte of the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 21.6.3.4, "Debug Store (DS) Mechanism."		If(CPUID.01H:EDX.DS[21] = 1
63:0	The linear address of the first byte of the DS buffer management area, if IA-32e mode is active.	
31:0	The linear address of the first byte of the DS buffer management area, if not in IA-32e mode.	
63:32	Reserved if not in IA-32e mode.	
Register Address: 6A0H, 1696		IA32_U_CET
Configure User Mode CET (R/W)		Bits 1:0 are defined if CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1. Bits 5:2 and bits 63:10 are defined if CPUID.(EAX=07H, ECX=0H):EDX.CET_IBT[20] = 1.
0	SH_STK_EN: When set to 1, enable shadow stacks at CPL3.	
1	WR_SHSTK_EN: When set to 1, enables the WRSSD/WRSSQ instructions.	
2	ENDBR_EN: When set to 1, enables indirect branch tracking.	
3	LEG_IW_EN: Enable legacy compatibility treatment for indirect branch tracking.	
4	NO_TRACK_EN: When set to 1, enables use of no-track prefix for indirect branch tracking.	
5	SUPPRESS_DIS: When set to 1, disables suppression of CET indirect branch tracking on legacy compatibility.	
9:6	Reserved; must be zero.	
10	SUPPRESS: When set to 1, indirect branch tracking is suppressed. This bit can be written to 1 only if TRACKER is written as IDLE.	
11	TRACKER: Value of the indirect branch tracking state machine. Values: IDLE (0), WAIT_FOR_ENDBRANCH(1).	
63:12	EB_LEG_BITMAP_BASE: Linear address bits 63:12 of a legacy code page bitmap used for legacy compatibility when indirect branch tracking is enabled. If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are used.	
Register Address: 6A2H, 1698		IA32_S_CET
Configure Supervisor Mode CET (R/W)		See IA32_U_CET (6A0H) for reference; similar format.
Register Address: 6A4H, 1700		IA32_PL0_SSP

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Linear address to be loaded into SSP on transition to privilege level 0. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. Bits 1:0 of the MSR must be 0. Transitions to privilege level 0 will check that bit 2 is also 0.		If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
Register Address: 6A5H, 1701		IA32_PL1_SSP
Linear address to be loaded into SSP on transition to privilege level 1. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. Bits 1:0 of the MSR must be 0. Transitions to privilege level 1 from a higher privilege level will check that bit 2 is also 0.		If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
Register Address: 6A6H, 1702		IA32_PL2_SSP
Linear address to be loaded into SSP on transition to privilege level 2. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. Bits 1:0 of the MSR must be 0. Transitions to privilege level 2 from a higher privilege level will check that bit 2 is also 0.		If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
Register Address: 6A7H, 1703		IA32_PL3_SSP
Linear address to be loaded into SSP on transition to privilege level 3. (R/W) If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are loaded. Bits 1:0 of the MSR must be 0.		If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
Register Address: 6A8H, 1704		IA32_INTERRUPT_SSP_TABLE_ADDR
Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W) This MSR is not present on processors that do not support Intel 64 architecture. This field cannot represent a non-canonical address.		If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1
Register Address: 6E0H, 1760		IA32_TSC_DEADLINE
TSC Target of Local APIC's TSC Deadline Mode (R/W)		If CPUID.01H:ECX.[24] = 1
63:0	REGISTER_VALUE TSC-deadline value.	
Register Address: 6E1H, 1761		IA32_PKRS
Specifies the PK permissions associated with each protection domain for supervisor pages (R/W)		If CPUID.(EAX=07H, ECX=0H):ECX.PKS [31] = 1
31:0	For domain i (i between 0 and 15), bits 2i and 2i+1 contain the AD and WD permissions, respectively.	
63:32	Reserved.	
Register Address: 770H, 1904		IA32_PM_ENABLE
Enable/disable HWP (R/W)		If CPUID.06H:EAX.[7] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
0	HWP_ENABLE (R/W) Note this bit can only be enabled once from the default value. Once set, writes to the HWP_ENABLE bit are ignored. Only RESET will clear this bit. Default = 0. See Section 16.4.2, "Enabling HWP."	If CPUID.06H:EAX.[7] = 1
63:1	Reserved.	
Register Address: 771H, 1905	IA32_HWP_CAPABILITIES	
HWP Performance Range Enumeration (R/O)		If CPUID.06H:EAX.[7] = 1
7:0	Highest_Performance See Section 16.4.3, "HWP Performance Range and Dynamic Capabilities."	If CPUID.06H:EAX.[7] = 1
15:8	Guaranteed_Performance See Section 16.4.3, "HWP Performance Range and Dynamic Capabilities."	If CPUID.06H:EAX.[7] = 1
23:16	Most_Efficient_Performance See Section 16.4.3, "HWP Performance Range and Dynamic Capabilities".	If CPUID.06H:EAX.[7] = 1
31:24	Lowest_Performance See Section 16.4.3, "HWP Performance Range and Dynamic Capabilities."	If CPUID.06H:EAX.[7] = 1
63:32	Reserved.	
Register Address: 772H, 1906	IA32_HWP_REQUEST_PKG	
Power Management Control Hints for All Logical Processors in a Package (R/W)		If CPUID.06H:EAX.[11] = 1
7:0	Minimum_Performance See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[11] = 1
15:8	Maximum_Performance See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[11] = 1
23:16	Desired_Performance See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[11] = 1
31:24	Energy_Performance_Preference See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[10] = 1
41:32	Activity_Window See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[9] = 1
63:42	Reserved.	
Register Address: 773H, 1907	IA32_HWP_INTERRUPT	
Control HWP Native Interrupts (R/W)		If CPUID.06H:EAX.[8] = 1
0	EN_Guaranteed_Performance_Change See Section 16.4.6, "HWP Notifications."	If CPUID.06H:EAX.[8] = 1
1	EN_Excursion_Minimum See Section 16.4.6, "HWP Notifications."	If CPUID.06H:EAX.[8] = 1
63:2	Reserved.	
Register Address: 774H, 1908	IA32_HWP_REQUEST	
Power Management Control Hints to a Logical Processor (R/W)		If CPUID.06H:EAX.[7] = 1
7:0	Minimum_Performance See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[7] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
15:8	Maximum_Performance See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[7] = 1
23:16	Desired_Performance See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[7] = 1
31:24	Energy_Performance_Preference See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[10] = 1
41:32	Activity_Window See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[9] = 1
42	Package_Control See Section 16.4.4, "Managing HWP."	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[11] = 1
63:43	Reserved.	
Register Address: 775H, 1909		IA32_PECI_HWP_REQUEST_INFO
IA32_PECI_HWP_REQUEST_INFO		
7:0	Minimum Performance (MINIMUM_PERFORMANCE): Used by OS to read the latest value of PECI minimum performance input. Default value is 0.	
15:8	Maximum Performance (MAXIMUM_PERFORMANCE): Used by OS to read the latest value of PECI maximum performance input. Default value is 0.	
23:16	Reserved.	
31:24	Energy Performance Preference (ENERGY_PERFORMANCE_PREFERENCE): Used by OS to read the latest value of PECI Energy Performance Preference input. Default value is 0.	
59:32	Reserved.	
60	EPP PECI Override (EPP_PECI_OVERRIDE): Indicates whether PECI is currently overriding the Energy Performance Preference input. If set to '1', PECI is overriding the Energy Performance Preference input. If clear (0), OS has control over Energy Performance Preference input. Default value is 0.	
61	Reserved.	
62	Max PECI Override (MAX_PECI_OVERRIDE): Indicates whether PECI is currently overriding the Maximum Performance input. If set to '1', PECI is overriding the Maximum Performance input. If clear (0), OS has control over Maximum Performance input. Default value is 0.	
63	Min PECI Override (MIN_PECI_OVERRIDE): Indicates whether PECI is currently overriding the Minimum Performance input. If set to '1', PECI is overriding the Minimum Performance input. If clear (0), OS has control over Minimum Performance input. Default value is 0.	
Register Address: 776H, 1910		IA32_HWP_CTL
IA32_HWP_CTL		If CPUID.06H:EAX.[22] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
0	PKG_CTL_POLARITY Defines which HWP Request MSR is used whether Thread level or package level. When package MSR is used, the thread MSR valid bits define which thread MSR fields override the package. Default value is 0.	If CPUID.06H:EAX.[22] = 1
63:1	Reserved.	
Register Address: 777H, 1911		IA32_HWP_STATUS
Log bits indicating changes to Guaranteed & excursions to Minimum (R/W)		If CPUID.06H:EAX.[7] = 1
0	Guaranteed_Performance_Change (R/WC0) See Section 16.4.5, "HWP Feedback."	If CPUID.06H:EAX.[7] = 1
1	Reserved.	
2	Excursion_To_Minimum (R/WC0) See Section 16.4.5, "HWP Feedback."	If CPUID.06H:EAX.[7] = 1
63:3	Reserved.	
Register Address: 7A3H, 1955		IA32_MCU_EXT_SERVICE
MCU Extended Service (R/O)		If IA32_ARCH_CAPABILITIES[22] = 1
3:0	ALLOWED_PERIODS Value indicates the allowed periods for extended servicing. Value x means that all extended servicing periods are allowed till period x.	
63:4	Reserved.	
Register Address: 7A4H, 1956		IA32_MCU_ROLLBACK_MIN_ID
Minimal MCU Revision ID (R/O) Minimal MCU Revision ID that software can rollback to per boot.		If IA32_MCU_ENUMERATION[3] = 1
31:0	REVISION_ID Minimal MCU revision ID for rollback.	
63:32	Reserved for future use.	
Register Address: 7A5H, 1957		IA32_MCU_STAGING_MBOX_ADDR
IA32_MCU_STAGING_MBOX_ADDR (R/O) Reports MMIO address of MCU staging DOE mailbox.		
63:0	ADDR MMIO address base of MCU staging DOE mailbox.	
Register Address: 7B0H, 1968		IA32_ROLLBACK_SIGN_ID_0
Rollback ID 0 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 7B1H, 1969	IA32_ROLLBACK_SIGN_ID_1	
Rollback ID 1 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7B2H, 1970	IA32_ROLLBACK_SIGN_ID_2	
Rollback ID 2 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7B3H, 1971	IA32_ROLLBACK_SIGN_ID_3	
Rollback ID 3 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7B4H, 1972	IA32_ROLLBACK_SIGN_ID_4	
Rollback ID 4 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7B5H, 1973	IA32_ROLLBACK_SIGN_ID_5	
Rollback ID 5 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63:48	Reserved.	
Register Address: 7B6H, 1974	IA32_ROLLBACK_SIGN_ID_6	
Rollback ID 6 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7B7H, 1975	IA32_ROLLBACK_SIGN_ID_7	
Rollback ID 7 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7B8H, 1976	IA32_ROLLBACK_SIGN_ID_8	
Rollback ID 8 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7B9H, 1977	IA32_ROLLBACK_SIGN_ID_9	
Rollback ID 9 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7BAH, 1978	IA32_ROLLBACK_SIGN_ID_10	
Rollback ID 10 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7BBH, 1979	IA32_ROLLBACK_SIGN_ID_11	
Rollback ID 11 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7BCH, 1980	IA32_ROLLBACK_SIGN_ID_12	
Rollback ID 12 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7BDH, 1981	IA32_ROLLBACK_SIGN_ID_13	
Rollback ID 13 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7BEH, 1982	IA32_ROLLBACK_SIGN_ID_14	
Rollback ID 14 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 7BFH, 1983	IA32_ROLLBACK_SIGN_ID_15	
Rollback ID 15 (R/O) Holds the Revision ID and SVN of a supported rollback target or 0 if none.		If IA32_MCU_ENUMERATION[3] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
31:0	MCU_ROLLBACK_ID MCU supported Rollback ID.	
47:32	ROLLBACK_MCU_SVN MCU SVN corresponding to the reported MCU Rollback ID.	
63:48	Reserved.	
Register Address: 802H, 2050	IA32_X2APIC_APICID	
x2APIC ID Register (R/O)		If CPUID.01H:ECX[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 803H, 2051	IA32_X2APIC_VERSION	
x2APIC Version Register (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 808H, 2056	IA32_X2APIC_TPR	
x2APIC Task Priority Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 80AH, 2058	IA32_X2APIC_PPR	
x2APIC Processor Priority Register (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 80BH, 2059	IA32_X2APIC_EOI	
x2APIC EOI Register (W/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 80DH, 2061	IA32_X2APIC_LDR	
x2APIC Logical Destination Register (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 80FH, 2063	IA32_X2APIC_SIVR	
x2APIC Spurious Interrupt Vector Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 810H, 2064	IA32_X2APIC_ISR0	
x2APIC In-Service Register Bits 31:0 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 811H, 2065	IA32_X2APIC_ISR1	
x2APIC In-Service Register Bits 63:32 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 812H, 2066	IA32_X2APIC_ISR2	
x2APIC In-Service Register Bits 95:64 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 813H, 2067	IA32_X2APIC_ISR3	
x2APIC In-Service Register Bits 127:96 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 814H, 2068	IA32_X2APIC_ISR4	
x2APIC In-Service Register Bits 159:128 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 815H, 2069	IA32_X2APIC_ISR5	
x2APIC In-Service Register Bits 191:160 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 816H, 2070	IA32_X2APIC_ISR6	
x2APIC In-Service Register Bits 223:192 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 817H, 2071	IA32_X2APIC_ISR7	
x2APIC In-Service Register Bits 255:224 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 818H, 2072	IA32_X2APIC_TMR0	
x2APIC Trigger Mode Register Bits 31:0 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 819H, 2073	IA32_X2APIC_TMR1	
x2APIC Trigger Mode Register Bits 63:32 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 81AH, 2074	IA32_X2APIC_TMR2	
x2APIC Trigger Mode Register Bits 95:64 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 81BH, 2075	IA32_X2APIC_TMR3	
x2APIC Trigger Mode Register Bits 127:96 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 81CH, 2076	IA32_X2APIC_TMR4	
x2APIC Trigger Mode Register Bits 159:128 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 81DH, 2077	IA32_X2APIC_TMR5	
x2APIC Trigger Mode Register Bits 191:160 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 81EH, 2078	IA32_X2APIC_TMR6	
x2APIC Trigger Mode Register Bits 223:192 (R/O)		If (CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1)
Register Address: 81FH, 2079	IA32_X2APIC_TMR7	
x2APIC Trigger Mode Register Bits 255:224 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 820H, 2080	IA32_X2APIC_IRR0	
x2APIC Interrupt Request Register Bits 31:0 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 821H, 2081	IA32_X2APIC_IRR1	
x2APIC Interrupt Request Register Bits 63:32 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 822H, 2082	IA32_X2APIC_IRR2	
x2APIC Interrupt Request Register Bits 95:64 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 823H, 2083	IA32_X2APIC_IRR3	
x2APIC Interrupt Request Register Bits 127:96 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 824H, 2084	IA32_X2APIC_IRR4	
x2APIC Interrupt Request Register Bits 159:128 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 825H, 2085	IA32_X2APIC_IRR5	
x2APIC Interrupt Request Register Bits 191:160 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 826H, 2086	IA32_X2APIC_IRR6	
x2APIC Interrupt Request Register Bits 223:192 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 827H, 2087	IA32_X2APIC_IRR7	
x2APIC Interrupt Request Register Bits 255:224 (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 828H, 2088	IA32_X2APIC_ESR	
x2APIC Error Status Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 82FH, 2095	IA32_X2APIC_LVT_CMCI	
x2APIC LVT Corrected Machine Check Interrupt Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 830H, 2096	IA32_X2APIC_ICR	
x2APIC Interrupt Command Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 832H, 2098	IA32_X2APIC_LVT_TIMER	
x2APIC LVT Timer Interrupt Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 833H, 2099	IA32_X2APIC_LVT_THERMAL	
x2APIC LVT Thermal Sensor Interrupt Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 834H, 2100	IA32_X2APIC_LVT_PMI	
x2APIC LVT Performance Monitor Interrupt Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 835H, 2101	IA32_X2APIC_LVT_LINT0	
x2APIC LVT LINT0 Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 836H, 2102	IA32_X2APIC_LVT_LINT1	
x2APIC LVT LINT1 Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 837H, 2103	IA32_X2APIC_LVT_ERROR	
x2APIC LVT Error Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 838H, 2104	IA32_X2APIC_INIT_COUNT	
x2APIC Initial Count Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 839H, 2105	IA32_X2APIC_CUR_COUNT	
x2APIC Current Count Register (R/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 83EH, 2110	IA32_X2APIC_DIV_CONF	
x2APIC Divide Configuration Register (R/W)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 83FH, 2111	IA32_X2APIC_SELF_IPI	
x2APIC Self IPI Register (W/O)		If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
Register Address: 981H, 2433	IA32_TME_CAPABILITY	
Memory Encryption Capability MSR		If CPUID.07H:ECX.[13] = 1
0	Support for AES-XTS 128-bit encryption algorithm. (NIST standard)	
1	Support for AES-XTS 128-bit encryption with integrity algorithm.	
2	Support for AES-XTS 256-bit encryption algorithm.	
29:3	Reserved.	
30	SUPPORT_IA32_TME_CLEAR_SAVED_KEY Support for the IA32_TME_CLEAR_SAVED_KEY MSR.	
31	TME encryption bypass supported.	
35:32	MK_TME_MAX_KEYID_BITS Number of bits which can be allocated for usage as key identifiers for multi-key memory encryption. 4 bits allow for a maximum value of 15, which could address 32K keys. Zero if TME-MK is not supported.	
50:36	MK_TME_MAX_KEYS Indicates the maximum number of keys which are available for usage. This value may not be a power of 2. KeyID 0 is specially reserved and is not accounted for in this field.	
63:51	Reserved.	
Register Address: 982H, 2434	IA32_TME_ACTIVATE	
Memory Encryption Activation MSR This MSR is used to lock the MSRs listed below. Any write to the following MSRs will be ignored after they are locked. The lock is reset when CPU is reset. • IA32_TME_ACTIVATE • IA32_TME_EXCLUDE_MASK • IA32_TME_EXCLUDE_BASE Note that IA32_TME_EXCLUDE_MASK and IA32_TME_EXCLUDE_BASE must be configured before IA32_TME_ACTIVATE.		If CPUID.07H:ECX.[13] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
0	Lock R/O – Will be set upon successful WRMSR (or first SMI); written value ignored.	
1	Hardware Encryption Enable This bit also enables TME-MK; TME-MK cannot be enabled without enabling encryption hardware.	
2	Key Select 0: Create a new TME key (expected cold/warm boot). 1: Restore the TME key from storage (Expected when resume from standby).	
3	Save TME Key for Standby Save key into storage to be used when resume from standby. Note: This may not be supported in all processors.	
7:4	TME Policy/Encryption Algorithm Only algorithms enumerated in IA32_TME_CAPABILITY are allowed. For example: 0000 – AES-XTS-128. 0001 – AES-XTS-128 with integrity. 0010 – AES-XTS-256. Other values are invalid.	
30:8	Reserved.	
31	TME Encryption Bypass Enable When encryption hardware is enabled: ▪ Total Memory Encryption is enabled using a CPU generated ephemeral key based on a hardware random number generator when this bit is set to 0. ▪ Total Memory Encryption is bypassed (no encryption/decryption for KeyID0) when this bit is set to 1. Software must inspect Hardware Encryption Enable (bit 1) and TME encryption bypass Enable (bit 31) to determine if TME encryption is enabled.	
35:32	MK_TME_KEYID_BITS Reserved if TME-MK is not enumerated, otherwise: The number of key identifier bits to allocate to TME-MK usage. Similar to enumeration, this is an encoded value. Writing a value greater than MK_TME_MAX_KEYID_BITS will result in #GP. Writing a non-zero value to this field will #GP if bit 1 of EAX (Hardware Encryption Enable) is not also set to ‘1, as encryption hardware must be enabled to use TME-MK. Example: To support 255 keys, this field would be set to a value of 8.	
47:36	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63:48	MK_TME_CRYPTO_ALGS Reserved if TME-MK is not enumerated, otherwise: Bit 48: AES-XTS 128. Bit 49: AES-XTS 128 with integrity. Bit 50: AES-XTS 256. Bit 63:51: Reserved (#GP) Bitmask for BIOS to set which encryption algorithms are allowed for TME-MK, would be later enforced by the key loading ISA ('1 = allowed).	
Register Address: 983H, 2435	IA32_TME_EXCLUDE_MASK	
Memory Encryption Exclude Mask		If CPUID.07H:ECX.[13] = 1
10:0	Reserved.	
11	Enable: When set to '1', then TME_EXCLUDE_BASE and TME_EXCLUDE_MASK are used to define an exclusion region for TME/TME-MK (for KeyID=0).	
MAXPHYADDR-1:12	TMEEMASK: This field indicates the bits that must match TMEEBASE in order to qualify as a TME/TME-MK (for KeyID=0) exclusion memory range access.	
63:MAXPHYADDR	Reserved; must be zero.	
Register Address: 984H, 2436	IA32_TME_EXCLUDE_BASE	
Memory Encryption Exclude Base		IF CPUID.07H:ECX.[13] = 1
11:0	Reserved.	
MAXPHYADDR-1:12	TMEEBASE: Base physical address to be excluded for TME/TME-MK (for KeyID=0) encryption.	
63:MAXPHYADDR	Reserved; must be zero.	
Register Address: 985H, 2437	IA32_UINTR_RR	
User Interrupt Request Register (R/W)		IF CPUID.07H.01H:EDX[13]=1
63:0	UIRR Bitmap of requested user interrupt vectors.	
Register Address: 986H, 2438	IA32_UINTR_HANDLER	
User Interrupt Handler Address (R/W)		IF CPUID.07H.01H:EDX[13]=1
63:0	UIHANDLER User interrupt handler linear address.	
Register Address: 987H, 2439	IA32_UINTR_STACKADJUST	
User Interrupt Stack Adjustment (R/W)		IF CPUID.07H.01H:EDX[13]=1
0	LOAD_RSP User interrupt stack mode.	
2:1	Reserved.	
63:3	STACK_ADJUST Stack adjust value.	
Register Address: 988H, 2440	IA32_UINTR_MISC	
User-Interrupt Target-Table Size and Notification Vector (R/W)		If CPUID.07H.01H:EDX[13]=1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
31:0	UITTSZ The highest index of a valid entry in the user-interrupt target table. Valid entries are indices 0..UITTSZ (inclusive).	
39:32	UINV User-interrupt notification vector.	
63:40	Reserved.	
Register Address: 989H, 2441	IA32_UINTR_PD	
User Interrupt PID Address (R/W)		If CPUID.07H.01H:EDX[13]=1
5:0	Reserved.	
63:6	UPIDADDR User-interrupt notification processing accesses a UPID at this linear address.	
Register Address: 98AH, 2442	IA32_UINTR_TT	
User-Interrupt Target Table (R/W)		If CPUID.07H.01H:EDX[13]=1
0	SENDUIPI_ENABLE User-interrupt target table is valid.	
3:1	Reserved.	
63:4	UITTADDR User-interrupt target table base linear address.	
Register Address: 990H, 2448	IA32_COPY_STATUS^5	
Status of Most Recent Platform to Local or Local to Platform Copies (R/O)		If ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] = 1))
0	IWKEY_COPY_SUCCESSFUL Status of most recent copy to or from IWKeyBackup.	If ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] = 1))
63:1	Reserved.	
Register Address: 991H, 2449	IA32_IWKEYBACKUP_STATUS^5	
Information about IWKeyBackup Register (R/O)		If ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] =1))
0	Backup/Restore Valid Cleared when a write to IWKeyBackup is initiated, and then set when the latest write of IWKeyBackup has been written to storage that persists across S3/S4 sleep state. If S3/S4 is entered between when an IWKeyBackup write occurs and when this bit is set, then IWKeyBackup may not be recovered after S3/S4 exit. During S3/S4 sleep state exit (system wake up), this bit is cleared. It is set again when IWKeyBackup is restored from persistent storage and thus available to be copied to IWKey using IA32_COPY_PLATFORM_TO_LOCAL MSR. Another write to IWKeyBackup (via IA32_COPY_LOCAL_TO_PLATFORM MSR) may fail if a previous write has not yet set this bit.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] =1))
1	Reserved.	
2	Backup Key Storage Read/Write Error Updated prior to backup/restore valid being set. Set when an error is encountered while backing up or restoring a key to persistent storage.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] =1))

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
3	IWKeyBackup Consumed Set after the previous backup operation has been consumed by the platform. This does not indicate that the system is ready for a second IWKeyBackup write as the previous IWKeyBackup write may still need to set Backup/restore valid.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(07H,0).ECX[23] =1))
63:4	Reserved.	
Register Address: 9FBH, 2555		IA32_TME_CLEAR_SAVED_KEY
IA32_TME_CLEAR_SAVED_KEY (W/O)		
0	TME_CLEAR_SAVED_KEY Clear saved TME keys.	
63:1	Reserved.	
Register Address: C80H, 3200		IA32_DEBUG_INTERFACE
Silicon Debug Feature Control (R/W)		If CPUID.01H:ECX.[11] = 1
0	Enable (R/W) BIOS set 1 to enable Silicon debug features. Default is 0.	If CPUID.01H:ECX.[11] = 1
29:1	Reserved.	
30	Lock (R/W): If 1, locks any further change to the MSR. The lock bit is set automatically on the first SMI assertion even if not explicitly set by BIOS. Default is 0.	If CPUID.01H:ECX.[11] = 1
31	Debug Occurred (R/O): This “sticky bit” is set by hardware to indicate the status of bit 0. Default is 0.	If CPUID.01H:ECX.[11] = 1
63:32	Reserved.	
Register Address: C81H, 3201		IA32_L3_QOS_CFG
L3 QOS Configuration (R/W)		If (CPUID.(EAX=10H, ECX=1):ECX.[2] = 1)
0	Enable (R/W) Set 1 to enable L3 CAT masks and CLOS to operate in Code and Data Prioritization (CDP) mode.	
63:1	Reserved. Attempts to write to reserved bits result in a #GP(0).	
Register Address: C82H, 3202		IA32_L2_QOS_CFG
L2 QOS Configuration (R/W)		If (CPUID.(EAX=10H, ECX=2):ECX.[2] = 1)
0	Enable (R/W) Set 1 to enable L2 CAT masks and CLOS to operate in Code and Data Prioritization (CDP) mode.	
63:1	Reserved. Attempts to write to reserved bits result in a #GP(0).	
Register Address: C83H, 3203		IA32_L3_IO_QOS_CFG
L3 I/O QOS Configuration (R/W) This MSR is used to enable the I/O RDT features.		If (CPUID.(EAX=0FH, ECX=1):EAX.[10:9] = 1)
0	L3 I/O RDT Allocation Enable.	
1	L3 I/O RDT Monitoring Enable.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63:2	Reserved.	
Register Address: C88H, 3208	IA32_RESOURCE_PRIORITY	
Thread scope Resource Priority Enable (R/W)		
0	ENABLE When set, enables model specific features that can be used to create a Resource Priority mode.	
63:1	Reserved.	
Register Address: C89H, 3209	IA32_RESOURCE_PRIORITY_PKG	
IA32_RESOURCE_PRIORITY_PKG (R/W)		
0	ENABLE Enable Resource Priority feature.	
63:1	Reserved.	
Register Address: C8DH, 3213	IA32_QM_EVTSEL	
Monitoring Event Select Register (R/W)		If (CPUID.(EAX=07H, ECX=0):EBX.[12] = 1)
7:0	Event ID: ID of a supported monitoring event to report via IA32_QM_CTR.	
31: 8	Reserved.	
N+31:32	Resource Monitoring ID: ID for monitoring hardware to report monitored data via IA32_QM_CTR.	N = Ceil (Log$_2$ (CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
63:N+32	Reserved.	
Register Address: C8EH, 3214	IA32_QM_CTR	
Monitoring Counter Register (R/O)		If (CPUID.(EAX=07H, ECX=0):EBX.[12] = 1)
61:0	Resource Monitored Data.	
62	Unavailable: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.	
63	Error: If 1, indicates an unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	
Register Address: C8FH, 3215	IA32_PQR_ASSOC	
Resource Association Register (R/W)		If ((CPUID.(EAX=07H, ECX=0):EBX[12] =1) or (CPUID.(EAX=07H, ECX=0):EBX[15] =1))
N-1:0	Resource Monitoring ID (R/W): ID for monitoring hardware to track internal operation, e.g., memory access.	N = Ceil (Log$_2$ (CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
31:N	Reserved.	
63:32	CLOS (R/W): The class of service (CLOS) to enforce (on writes); returns the current CLOS when read.	If (CPUID.(EAX=07H, ECX=0):EBX.[15] = 1)
Register Address: C90H—D8FH, 3216—3471	Reserved MSR Address Space for CAT Mask Registers	
See Section 19.19.4.1, “Enumeration and Detection Support of Cache Allocation Technology.”		
Register Address: C90H, 3216	IA32_L3_MASK_0	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
L3 CAT Mask for COS0 (R/W)		If (CPUID.(EAX=10H, ECX=0H):EBX[1] != 0)
31:0	Capacity Bit Mask (R/W)	
63:32	Reserved.	
Register Address: C90H+n, 3216+n	IA32_L3_MASK_n	
L3 CAT Mask for COSn (R/W)		n = CPUID.(EAX=10H, ECX=1H):EDX[15:0]
31:0	Capacity Bit Mask (R/W)	
63:32	Reserved.	
Register Address: D10H—D4FH, 3344—3407	Reserved MSR Address Space for L2 CAT Mask Registers	
See Section 19.19.4.1, “Enumeration and Detection Support of Cache Allocation Technology.”		
Register Address: D10H, 3344	IA32_L2_MASK_0	
L2 CAT Mask for COS0 (R/W)		If (CPUID.(EAX=10H, ECX=0H):EBX[2] != 0)
31:0	Capacity Bit Mask (R/W)	
63:32	Reserved.	
Register Address: D10H+n, 3344+n	IA32_L2_MASK_n	
L2 CAT Mask for COSn (R/W)		n = CPUID.(EAX=10H, ECX=2H):EDX[15:0]
31:0	Capacity Bit Mask (R/W)	
63:32	Reserved.	
Register Address: D18H, 3352	IA32_L2_MASK_8	
L2 CAT Mask for COS8 (R/W)		
15:0	WAY_MASK Capacity Bit Mask. Available ways vectors for class of service of IA core. ’1 in bit indicates allocation to the way is allowed. ’0 indicates allocation to the way is not allowed.	
63:16	Reserved.	
Register Address: D19H, 3353	IA32_L2_MASK_9	
L2 CAT Mask for COS9 (R/W) See IA32_L2_MASK_8 (D18H) for reference; similar format.		
Register Address: D1AH, 3354	IA32_L2_MASK_10	
L2 CAT Mask for COS10 (R/W) See IA32_L2_MASK_8 (D18H) for reference; similar format.		
Register Address: D1BH, 3355	IA32_L2_MASK_11	
L2 CAT Mask for COS11 (R/W) See IA32_L2_MASK_8 (D18H) for reference; similar format.		
Register Address: D1CH, 3356	IA32_L2_MASK_12	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
L2 CAT Mask for COS12 (R/W) See IA32_L2_MASK_8 (D18H) for reference; similar format.		
Register Address: D1DH, 3357		IA32_L2_MASK_13
L2 CAT Mask for COS13 (R/W) See IA32_L2_MASK_8 (D18H) for reference; similar format.		
Register Address: D1EH, 3358		IA32_L2_MASK_14
L2 CAT Mask for COS14 (R/W) See IA32_L2_MASK_8 (D18H) for reference; similar format.		
Register Address: D1FH, 3359		IA32_L2_MASK_15
L2 CAT Mask for COS15 (R/W) See IA32_L2_MASK_8 (D18H) for reference; similar format.		
Register Address: D50H, 3408		IA32_L2_QOS_EXT_BW_THRTL_0
IA32_L2_QOS_EXT_BW_THRTL_0 (R/W) Memory Bandwidth enforcement for COS0.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 0
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D51H, 3409		IA32_L2_QOS_EXT_BW_THRTL_1
IA32_L2_QOS_EXT_BW_THRTL_1 (R/W) Memory Bandwidth enforcement for COS1.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 1
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D52H, 3410		IA32_L2_QOS_EXT_BW_THRTL_2
IA32_L2_QOS_EXT_BW_THRTL_2 (R/W) Memory Bandwidth enforcement for COS2.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 2
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D53H, 3411		IA32_L2_QOS_EXT_BW_THRTL_3
IA32_L2_QOS_EXT_BW_THRTL_3 (R/W) Memory Bandwidth enforcement for COS3.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 3
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63:7	Reserved.	
Register Address: D54H, 3412	IA32_L2_QOS_EXT_BW_THRTL_4	
IA32_L2_QOS_EXT_BW_THRTL_4 (R/W) Memory Bandwidth enforcement for COS4.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 4
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D55H, 3413	IA32_L2_QOS_EXT_BW_THRTL_5	
IA32_L2_QOS_EXT_BW_THRTL_5 (R/W) Memory Bandwidth enforcement for COS5.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 5
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D56H, 3414	IA32_L2_QOS_EXT_BW_THRTL_6	
IA32_L2_QOS_EXT_BW_THRTL_6 (R/W) Memory Bandwidth enforcement for COS6.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 6
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D57H, 3415	IA32_L2_QOS_EXT_BW_THRTL_7	
IA32_L2_QOS_EXT_BW_THRTL_7 (R/W) Memory Bandwidth enforcement for COS7.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 7
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D58H, 3416	IA32_L2_QOS_EXT_BW_THRTL_8	
IA32_L2_QOS_EXT_BW_THRTL_8 (R/W) Memory Bandwidth enforcement for COS8.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 8
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D59H, 3417	IA32_L2_QOS_EXT_BW_THRTL_9	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
IA32_L2_QOS_EXT_BW_THRTL_9 (R/W) Memory Bandwidth enforcement for COS9.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 9
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D5AH, 3418	IA32_L2_QOS_EXT_BW_THRTL_10	
IA32_L2_QOS_EXT_BW_THRTL_10 (R/W) Memory Bandwidth enforcement for COS10.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 10
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D5BH, 3419	IA32_L2_QOS_EXT_BW_THRTL_11	
IA32_L2_QOS_EXT_BW_THRTL_11 (R/W) Memory Bandwidth enforcement for COS11.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 11
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D5CH, 3420	IA32_L2_QOS_EXT_BW_THRTL_12	
IA32_L2_QOS_EXT_BW_THRTL_12 (R/W) Memory Bandwidth enforcement for COS12.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 12
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D5DH, 3421	IA32_L2_QOS_EXT_BW_THRTL_13	
IA32_L2_QOS_EXT_BW_THRTL_13 (R/W) Memory Bandwidth enforcement for COS13.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 13
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D5EH, 3422	IA32_L2_QOS_EXT_BW_THRTL_14	
IA32_L2_QOS_EXT_BW_THRTL_14 (R/W) Memory Bandwidth enforcement for COS14.		CPUID.(EAX=10H,ECX=0H):EBX[3] and CPUID.(EAX=10H,ECX=3H):EDX ≥ 14

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
6:0	RBE_ENFORCEMENT_VAL Max Delay value cannot be greater than 90 percent - 0x5a.	
63:7	Reserved.	
Register Address: D90H, 3472		IA32_BNDCFGS
Supervisor State of MPX Configuration (R/W)		If (CPUID.(EAX=07H, ECX=0H):EBX[14] = 1)
0	EN: Enable Intel MPX in supervisor mode.	
1	BNDPRESERVE: Preserve the bounds registers for near branch instructions in the absence of the BND prefix.	
11:2	Reserved, must be zero.	
63:12	Base Address of Bound Directory.	
Register Address: D91H, 3473		IA32_COPY_LOCAL_TO_PLATFORM[5]
Copy Local State to Platform State (W)		IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H).ECX[23] = 1))
0	IWKeyBackup Copy IWKey to IWKeyBackup.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H).ECX[23] = 1))
63:1	Reserved.	
Register Address: D92H, 3474		IA32_COPY_PLATFORM_TO_LOCAL[5]
Copy Platform State to Local State (W)		IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H).ECX[23] = 1))
0	IWKeyBackup Copy IWKeyBackup to IWKey.	IF ((CPUID.19H:EBX[4] = 1) && (CPUID.(EAX=07H, ECX=0H).ECX[23] = 1))
63:1	Reserved.	
Register Address: D93H, 3475		IA32_PASID
Process Address Space Identifier. (R/W)		
19:0	Process address space identifier (PASID). Specifies the PASID of the currently running software thread.	
30:20	Reserved.	
31	Valid. Execution of ENQCMD causes a #GP if this bit is clear.	
63:32	Reserved.	
Register Address: DA0H, 3488		IA32_XSS
Extended Supervisor State Mask (R/W)		If(CPUID.(0DH, 1):EAX.[3] = 1
7:0	Reserved.	
8	PT State (R/W)	
9	Reserved.	
10	PASID State (R/W)	
11	CET_U State (R/W)	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
12	CET_S State (R/W)	
13	HDC State (R/W)	
14	UINTR State (R/W)	
15	LBR State (R/W)	
16	HWP State (R/W)	
63:17	Reserved.	
Register Address: DB0H, 3504	IA32_PKG_HDC_CTL	
Package Level Enable/Disable HDC (R/W)		If CPUID.06H:EAX.[13] = 1
0	HDC_Pkg_Enable (R/W) Force HDC idling or wake up HDC-idled logical processors in the package. See Section 16.5.2, “Package level Enabling HDC.”	If CPUID.06H:EAX.[13] = 1
63:1	Reserved.	
Register Address: DB1H, 3505	IA32_PM_CTL1	
Enable/Disable the HDC Thread Level Activity (R/W)		If CPUID.06H:EAX.[13] = 1
0	SDC_ALLOWED (R/W) Set this bit to allow this thread to be forced into HDC idle state. Clearing this bit blocks HDC-enter (HW) request. Default value: 1. See Section 16.5.3.	If CPUID.06H:EAX.[13] = 1
63:1	Reserved.	
Register Address: DB2H, 3506	IA32_THREAD_STALL	
Per-Logical_Processor_ID HDC Idle Residency (R/O)		If CPUID.06H:EAX.[13] = 1
63:0	Stall_Cycle_Cnt (R/W) Stalled cycles due to HDC forced idle on this logical processor. See Section 16.5.4.1.	If CPUID.06H:EAX.[13] = 1
Register Address: E00H, 3584	IA32_QOS_CORE_BW_THRTL_0	
CBA Levels Based on COS for Bandwidth Throttling (R/W)		CPUID.10H.0H:EBX[5]=1
3:0	COS0_LEVEL CBA Level for COS[0]. Levels are programmed from 0 to 15.	
7:4	Reserved.	
11:8	COS1_LEVEL CBA Level for COS[1]. Levels are programmed from 0 to 15.	
15:12	Reserved.	
19:16	COS2_LEVEL CBA Level for COS[2]. Levels are programmed from 0 to 15.	
25:20	Reserved.	
27:24	COS3_LEVEL CBA Level for COS[3]. Levels are programmed from 0 to 15.	
31:28	Reserved.	
35:32	COS4_LEVEL CBA Level for COS[4]. Levels are programmed from 0 to 15.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
39:36	Reserved.	
43:40	COS5_LEVEL CBA Level for COS[5]. Levels are programmed from 0 to 15.	
47:44	Reserved.	
51:48	COS6_LEVEL CBA Level for COS[6]. Levels are programmed from 0 to 15.	
Register Address: E01H, 3585		IA32_QOS_CORE_BW_THRTL_1
CBA Levels Based on COS for Bandwidth Throttling (R/W)		CPUID.10H.0H:EBX[5]=1
3:0	COS8_LEVEL CBA Level for COS[8]. Levels are programmed from 0 to 15.	
7:4	Reserved.	
11:8	COS9_LEVEL CBA Level for COS[9]. Levels are programmed from 0 to 15.	
15:12	Reserved.	
19:16	COS10_LEVEL CBA Level for COS[10]. Levels are programmed from 0 to 15.	
25:20	Reserved.	
27:24	COS11_LEVEL CBA Level for COS[11]. Levels are programmed from 0 to 15.	
31:28	Reserved.	
35:32	COS12_LEVEL CBA Level for COS[12]. Levels are programmed from 0 to 15.	
39:36	Reserved.	
43:40	COS13_LEVEL CBA Level for COS[13]. Levels are programmed from 0 to 15.	
47:44	Reserved.	
51:48	COS14_LEVEL CBA Level for COS[14]. Levels are programmed from 0 to 15.	
55:50	Reserved.	
59:56	COS15_LEVEL CBA Level for COS[15]. Levels are programmed from 0 to 15.	
63:60	Reserved	
Register Address: 1200H—121FH, 4608—4639		IA32_LBR_x_INFO
Last Branch Record Entry X Info Register (R/W) An attempt to read or write IA32_LBR_x_INFO such that x ≥ IA32_LBR_DEPTH.DEPTH will #GP.		
15:0	CYC_CNT The elapsed CPU cycles (saturating) since the last LBR was recorded. See Section 18.1.3.3.	Reset Value: 0
55:16	Undefined, may be zero or non-zero. Writes of non- zero values do not fault, but reads may return a different value.	Reset Value: 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
59:56	BR_TYPE The branch type recorded by this LBR. Encodings: 0000B: COND 0001B: JMP Indirect 0010B: JMP Direct 0011B: CALL Indirect 0100B: CALL Direct 0101B: RET 011xB: Reserved 1xxxB: Other Branch	Reset Value: 0
60	CYC_CNT_VALID CYC_CNT value is valid. See Section 20.1.3.3.	Reset Value: 0
61	TSX_ABORT This LBR record is a TSX abort. On processors that do not support Intel TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.	Reset Value: 0
62	IN_TSX This LBR record records a branch that retired during a TSX transaction. On processors that do not support Intel TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.	Reset Value: 0
63	MISPRED The recorded branch direction (conditional branch) or target (indirect branch) was mispredicted.	Reset Value: 0
Register Address: 1400H, 5120		IA32_SEAMRR_BASE
SEAM Memory Range Register for TDX - Base Address (R/W)		
2:0	Reserved.	
3	CONFIGURED Set to 1 by BIOS if range is configured.	
24:4	Reserved.	
51:25	BASE SEAM Range Register BASE address.	
63:52	Reserved.	
Register Address: 1401H, 5121		IA32_SEAMRR_MASK
SEAM Memory Range Register for TDX (R/W)		
9:0	Reserved.	
10	LOCK Set by BIOS to indicate range is configured and locked.	
24:11	Reserved.	
51:25	MASK Mask value for SEAMRR matching. Lowest granularity is 32M.	
63:52	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Register Address: 1406H, 5126	IA32_MCU_CONTROL	
MCU Control (R/W) Controls the behavior of the Microcode Update Trigger MSR, IA32_BIOS_UPDT_TRIG.		If CPUID.07H.0H:EDX[29]=1 && IA32_ARCH_CAPABILITIES.MCU_CONTROL=1
0	LOCK Once set, further writes to this MSR will cause a #GP(0) fault. Bypassed during SMM if EN_SMM_BYPASS (bit 2) is set.	
1	DIS_MCU_LOAD If this bit is set on a given logical processor, then any subsequent attempts to load a microcode update by that logical processor will be silently dropped (WRMSR 0x79 has no effect).	
2	EN_SMM_BYPASS If set, then writes to IA32_MCU_CONTROL are allowed during SMM regardless of the LOCK bit. This enables BIOS to Opt-In to the SMM Bypass functionality.	
63:3	Reserved.	
Register Address: 14CEH, 5326	IA32_LBR_CTL	
Last Branch Record Enabling and Configuration Register (R/W)		
0	LBREn When set, enables LBR recording.	Reset Value: 0
1	OS When set, allows LBR recording when CPL == 0.	Reset Value: 0
2	USR When set, allows LBR recording when CPL != 0.	Reset Value: 0
3	CALL_STACK When set, records branches in call-stack mode. See Section 20.1.2.4.	Reset Value: 0
15:4	Reserved.	Reset Value: 0
16	COND When set, records taken conditional branches. See Section 20.1.2.3.	
17	NEAR_REL_JMP When set, records near relative JMPs. See Section 20.1.2.3.	
18	NEAR_IND_JMP When set, records near indirect JMPs. See Section 20.1.2.3.	
19	NEAR_REL_CALL When set, records near relative CALLs. See Section 20.1.2.3.	
20	NEAR_IND_CALL When set, records near indirect CALLs. See Section 20.1.2.3.	
21	NEAR_RET When set, records near RETs. See Section 20.1.2.3.	
22	OTHER_BRANCH When set, records other branches. See Section 20.1.2.3.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63:23	Reserved.	
Register Address: 14CFH, 5327	IA32_LBR_DEPTH	
Last Branch Record Maximum Stack Depth Register (R/W)		
N:0	DEPTH The number of LBRs to be used for recording. Supported values are indicated by the bitmap in CPUID.(EAX=01CH,ECX=0):EAX[7:0]. The reset value will match the maximum supported by the CPU. Writes of unsupported values will #GP fault.	Reset Value: Varies
63:N+1	Reserved.	Reset Value: 0
Register Address: 1500H—151FH, 5376—5407	IA32_LBR_x_FROM_IP	
Last Branch Record entry X source IP register (R/W). An attempt to read or write IA32_LBR_x_FROM_IP such that x ≥ IA32_LBR_DEPTH.DEPTH will #GP.		
63:0	FROM_IP The source IP of the recorded branch or event, in canonical form. Writes to bits above MAXLINADDR-1 are ignored.	Reset Value: 0
Register Address: 1600H—161FH, 5632—5663	IA32_LBR_x_TO_IP	
Last Branch Record Entry X Destination IP Register (R/W) An attempt to read or write IA32_LBR_x_TO_IP such that x ≥ IA32_LBR_DEPTH.DEPTH will #GP.		
63:0	TO_IP The destination IP of the recorded branch or event, in canonical form. Writes to bits above MAXLINADDR-1 are ignored.	Reset Value: 0
Register Address: 17D0H, 6096	IA32_HW_FEEDBACK_PTR	
Hardware Feedback Interface Pointer		If CPUID.06H:EAX.[19] = 1
0	Valid (R/W) When set to 1, indicates a valid pointer is programmed into the ADDR field of the MSR.	
11:1	Reserved.	
(MAXPHYADDR-1):12	ADDR (R/W) Physical address of the page frame of the first page of the hardware feedback interface structure.	
63:MAXPHYADDR	Reserved.	
Register Address: 17D1H, 6097	IA32_HW_FEEDBACK_CONFIG	
Hardware Feedback Interface Configuration		If CPUID.06H:EAX.[19] = 1
0	Enable (R/W) When set to 1, enables the hardware feedback interface.	
63:1	Reserved.	
Register Address: 17D2H, 6098	IA32_THREAD_FEEDBACK_CHAR	
Thread Feedback Characteristics (R/O)		If CPUID.06H:EAX.[23] = 1
7:0	Application Class ID, pointing into the Intel Thread Director structure.	
62:8	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63	Valid bit. When set to 1 the OS Scheduler can use the Class ID (in bits 7:0) for its scheduling decisions. If this bit is 0, the Class ID field should be ignored. It is recommended that the OS uses the last known Class ID of the software thread for its scheduling decisions.	
Register Address: 17D4H, 6100		IA32_HW_FEEDBACK_THREAD_CONFIG
Hardware Feedback Thread Configuration (R/W)		
0	Enables Intel Thread Director. When set to 1, logical processor scope Intel Thread Director is enabled. Default is 0 (disabled).	
63:1	Reserved.	
Register Address: 17DAH, 6106		IA32_HRESET_ENABLE
History Reset Enable (R/W)		
0	Enable reset of the Intel Thread Director history.	
31:1	Reserved for other capabilities that can be reset by the HRESET instruction.	
63:32	Reserved.	
Register Address: 1900H, 6400		IA32_PMC_GP0_CTR
Full Width Writable General Performance Counter 0 (R/W)		If CPUID.0AH:EAX[15:8] > 0 and IA32_PERF_CAPABILITIES[13] =1
47:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:48	Reserved.	
Register Address: 1901H, 6401		IA32_PMC_GP0_CFG_A
IA32_PMC_GP0_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 0.		If CPUID.0AH:EAX[15:8] > 0
7:0	EVENT_SELECT Selects a performance event logic unit.	
15:8	UMASK Qualifies the microarchitectural condition to detect on the selected event logic.	
16	USR When set, events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.	
17	OS When set, events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USER flag.	
18	EDGE When set, enables edge detection of events.	
19	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
20	INT When set, the processor generates an exception through its local APIC on counter overflow for this counter's thread.	
21	ANYTHREAD If CPUID.A0H.EDX[15] is 1, then this bit is deprecated. When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	
22	ENABLE When set, performance counting is enabled in the performance-monitoring counter; when clear, the counter is disabled.	
23	INVERT Inverts the result of the counter-mask (CMASK) comparison when set, so that both greater than equal to and less than comparisons can be made. 0: The comparison is: threshold is greater than or equal to the event 1: The comparison is inverted: threshold is less than event.	
31:24	CMASK When CMASK is not zero, the corresponding performance counter increments by 1 each cycle if the event count is >= CMASK. This mask enables counting cycles in which multiple occurrences happen (for example, two or more instructions retired per clock).	
34:32	Reserved.	
35	EN_LBR_LOG When set enables updating LBRs with that counters event occurrences, if selected event is precise.	
36	EQUAL When EQ flag is set and the INV flag is clear, the comparison evaluates to true if the selected performance monitoring event (the event) is equal to the specified Counter Mask value (CMask). When EQ flag is set and INV flag is set, the comparison evaluates to true if the event is less-than the CMask value and the event is not zero. Note if CMask is zero, the EQ flag is ignored.	
39:37	Reserved.	
47:40	UMASK2 Unit mask 2 (UMASK2) field (bits 40 through 47) - These bits qualify the condition that the selected event logic unit detects. Valid UMASK2 values for each event logic unit are specific to the unit. The new UMASK2 field may also be used in conjunction with UMASK.	
63:48	Reserved.	
Register Address: 1903H, 6403		IA32_PMC_GP0_CFG_C
IA32_PMC_GP0_CFG_C (R/W) Extended Perf event selector for GP counter 0.		

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
31:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:32	Reserved.	
Register Address: 1904H, 6404		IA32_PMC_GP1_CTR
Full Width Writable General Performance Counter 1 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 1 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 1905H, 6405		IA32_PMC_GP1_CFG_A
IA32_PMC_GP1_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 1. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 1
Register Address: 1907H, 6407		IA32_PMC_GP1_CFG_C
IA32_PMC_GP1_CFG_C (R/W) Extended Perf event selector for GP counter 1. See IA32_PMC_GP0_CFG_C (1903H) for reference; similar format.		
Register Address: 1908H, 6408		IA32_PMC_GP2_CTR
Full Width Writable General Performance Counter 2 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 2 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 1909H, 6409		IA32_PMC_GP2_CFG_A
IA32_PMC_GP2_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 2. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 2
Register Address: 190AH, 6410		IA32_PMC_GP2_CFG_B
IA32_PMC_GP2_CFG_B (R/W) GP counter reload configuration register.		
1:0	Reserved.	
2	RELOAD_PMC2 Reload GP2 when GP2 overflows.	
3	RELOAD_PMC3 Reload GP2 when GP3 overflows.	
4	RELOAD_PMC4 Reload GP2 when GP4 overflows.	
5	RELOAD_PMC5 Reload GP2 when GP5 overflows.	
6	RELOAD_PMC6 Reload GP2 when GP6 overflows.	
7	RELOAD_PMC7 Reload GP2 when GP7 overflows.	
31:8	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
32	RELOAD_FC0 Reload GP2 when FC0 overflows.	
33	RELOAD_FC1 Reload GP2 when FC1 overflows.	
47:34	Reserved.	
48	METRICS_CLEAR Clear PERF_METRICS on overflow of GP2.	
63:49	Reserved.	
Register Address: 190BH, 6411	IA32_PMC_GP2_CFG_C	
IA32_PMC_GP2_CFG_C (R/W) Extended Perf event selector for GP counter 2. See IA32_PMC_GP0_CFG_C (1903H) for reference; similar format.		
Register Address: 190CH, 6412	IA32_PMC_GP3_CTR	
Full Width Writable General Performance Counter 3 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 3 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 190DH, 6413	IA32_PMC_GP3_CFG_A	
IA32_PMC_GP3_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 3. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 3
Register Address: 190EH, 6414	IA32_PMC_GP3_CFG_B	
IA32_PMC_GP3_CFG_B (R/W) GP counter reload configuration register. See IA32_PMC_GP2_CFG_B (190AH) for reference; similar format.		
Register Address: 190FH, 6415	IA32_PMC_GP3_CFG_C	
IA32_PMC_GP3_CFG_C (R/W) Extended Perf event selector for GP counter 3. See IA32_PMC_GP0_CFG_C (1903H) for reference; similar format.		
Register Address: 1910H, 6416	IA32_PMC_GP4_CTR	
Full Width Writable General Performance Counter 4 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 4 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 1911H, 6417	IA32_PMC_GP4_CFG_A	
IA32_PMC_GP4_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 4. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 4
Register Address: 1912H, 6418	IA32_PMC_GP4_CFG_B	
IA32_PMC_GP4_CFG_B (R/W) GP counter reload configuration register. See IA32_PMC_GP2_CFG_B (190AH) for reference; similar format.		
Register Address: 1913H, 6419	IA32_PMC_GP4_CFG_C	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
IA32_PMC_GP4_CFG_C (R/W) Extended Perf event selector for GP counter 4. See IA32_PMC_GP0_CFG_C (1903H) for reference; similar format.		
Register Address: 1914H, 6420		IA32_PMC_GP5_CTR
Full Width Writable General Performance Counter 5 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 5 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 1915H, 6421		IA32_PMC_GP5_CFG_A
IA32_PMC_GP5_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 5. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 5
Register Address: 1916H, 6422		IA32_PMC_GP5_CFG_B
IA32_PMC_GP5_CFG_B (R/W) GP counter reload configuration register. See IA32_PMC_GP2_CFG_B (190AH) for reference; similar format.		
Register Address: 1917H, 6423		IA32_PMC_GP5_CFG_C
IA32_PMC_GP5_CFG_C (R/W) Extended Perf event selector for GP counter 5. See IA32_PMC_GP0_CFG_C (1903H) for reference; similar format.		
Register Address: 1918H, 6424		IA32_PMC_GP6_CTR
Full Width Writable General Performance Counter 6 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 6 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 1919H, 6425		IA32_PMC_GP6_CFG_A
IA32_PMC_GP6_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 6. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 6
Register Address: 191AH, 6426		IA32_PMC_GP6_CFG_B
IA32_PMC_GP6_CFG_B (R/W) GP counter reload configuration register. See IA32_PMC_GP2_CFG_B (190AH) for reference; similar format.		
Register Address: 191BH, 6427		IA32_PMC_GP6_CFG_C
IA32_PMC_GP6_CFG_C (R/W) Extended Perf event selector for GP counter 6. See IA32_PMC_GP0_CFG_C (1903H) for reference; similar format.		
Register Address: 191CH, 6428		IA32_PMC_GP7_CTR
Full Width Writable General Performance Counter 7 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 7 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 191DH, 6429		IA32_PMC_GP7_CFG_A

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
IA32_PMC_GP7_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 7. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 7
Register Address: 191EH, 6430		IA32_PMC_GP7_CFG_B
IA32_PMC_GP7_CFG_B (R/W) GP counter reload configuration register. See IA32_PMC_GP2_CFG_B (190AH) for reference; similar format.		
Register Address: 191FH, 6431		IA32_PMC_GP7_CFG_C
IA32_PMC_GP7_CFG_C (R/W) Extended Perf event selector for GP counter 7. See IA32_PMC_GP0_CFG_C (1903H) for reference; similar format.		
Register Address: 1920H, 6432		IA32_PMC_GP8_CTR
Full Width Writable General Performance Counter 8 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 8 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 1921H, 6433		IA32_PMC_GP8_CFG_A
IA32_PMC_GP8_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 8. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 8
Register Address: 1924H, 6436		IA32_PMC_GP9_CTR
Full Width Writable General Performance Counter 9 (R/W) See IA32_PMC_GP0_CTR (1900H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 9 and IA32_PERF_CAPABILITIES[13]=1
Register Address: 1925H, 6437		IA32_PMC_GP9_CFG_A
IA32_PMC_GP9_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 9. See IA32_PMC_GP0_CFG_A (1901H) for reference; similar format.		If CPUID.0AH:EAX[15:8] > 9
Register Address: 1980H, 6528		IA32_PMC_FX0_CTR
Fixed-Function Performance Counter 0 (R/W) Instructions retired.		If CPUID.0AH:EDX[4:0] >0
47:0	FIXED_COUNTER Instructions Retired Counter.	
63:46	Reserved.	
Register Address: 1982H, 6530		IA32_PMC_FX0_CFG_B
Fixed-Function Counter Reload Configuration Register (R/W)		
1:0	Reserved.	
2	RELOAD_PMC2 Reload Fixed-Function Counter0 when GP2 overflows.	
3	RELOAD_PMC3 Reload Fixed-Function Counter0 when GP3 overflows.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
4	RELOAD_PMC4 Reload Fixed-Function Counter0 when GP4 overflows.	
5	RELOAD_PMC5 Reload Fixed-Function Counter0 when GP5overflows.	
6	RELOAD_PMC6 Reload Fixed-Function Counter0 when GP6 overflows.	
7	RELOAD_PMC7 Reload Fixed-Function Counter0 when GP7 overflows.	
33:8	Reserved.	
32	RELOAD_FC0 Reload Fixed-Function Counter0 when FC0 overflows.	
33	RELOAD_FC1 Reload Fixed-Function Counter0 when FC1 overflows.	
47:34	Reserved.	
48	METRICS_CLEAR Clear PERF_METRICS on overflow of Fixed-Function Counter 0.	
63:49	Reserved.	
Register Address: 1983H, 6531	IA32_PMC_FX0_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 0 (R/W)		
31:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:32	Reserved.	
Register Address: 1984H, 6532	IA32_PMC_FX1_CTR	
Fixed-Function Performance Counter 1 (R/W) Unhalted core clock cycles.		If CPUID.0AH:EDX[4:0] >1
47:0	FIXED_COUNTER Unhalted core clock cycles counter.	
63:46	Reserved.	
Register Address: 1986H, 6534	IA32_PMC_FX1_CFG_B	
Fixed-Function Counter Reload Configuration Register (R/W)		
1:0	Reserved.	
2	RELOAD_PMC2 Reload Fixed-Function Counter1 when GP2 overflows.	
3	RELOAD_PMC3 Reload Fixed-Function Counter1 when GP3 overflows.	
4	RELOAD_PMC4 Reload Fixed-Function Counter1 when GP4 overflows.	
5	RELOAD_PMC5 Reload Fixed-Function Counter1 when GP5overflows.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
6	RELOAD_PMC6 Reload Fixed-Function Counter1 when GP6 overflows.	
7	RELOAD_PMC7 Reload Fixed-Function Counter1 when GP7 overflows.	
31:8	Reserved.	
32	RELOAD_FC0 Reload Fixed-Function Counter1 when FC0 overflows.	
33	RELOAD_FC1 Reload Fixed-Function Counter1 when FC1 overflows.	
47:34	Reserved.	
48	METRICS_CLEAR Clear PERF_METRICS on overflow of Fixed-Function Counter 1.	
63:49	Reserved.	
Register Address: 1987H, 6532	IA32_PMC_FX1_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 1 (R/W)		
31:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:32	Reserved.	
Register Address: 1988H, 6536	IA32_PMC_FX2_CTR	
Fixed-Function Performance Counter 2 (R/W) Unhalted core reference cycles.		If CPUID.0AH:EDX[4:0] >2
47:0	FIXED_COUNTER Unhalted core reference cycles counter.	
63:48	Reserved.	
Register Address: 198BH, 6539	IA32_PMC_FX2_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 2 (R/W)		
31:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:32	Reserved.	
Register Address: 198CH, 6540	IA32_PMC_FX3_CTR	
Fixed-Function Performance Counter 3 (R/W) Top-down Microarchitecture Analysis unhalted number of available slots.		If CPUID.0AH:EDX[4:0] >3
47:0	FIXED_COUNTER Top-down microarchitecture analysis unhalted number of available slots counter.	
63:48	Reserved.	
Register Address: 1990H, 6544	IA32_PMC_FX4_CTR	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
Fixed-Function Performance Counter 4 (R/W) Top-down bad speculation.		If CPUID.0AH:EDX[4:0] >4
47:0	FIXED_COUNTER Top-down bad speculation counter.	
63:48	Reserved.	
Register Address: 1993H, 6547	IA32_PMC_FX4_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 4 (R/W)		
31:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:32	Reserved.	
Register Address: 1994H, 6548	IA32_PMC_FX5_CTR	
Fixed-Function Performance Counter 5 (R/W) Top-down frontend bound.		If CPUID.0AH:EDX[4:0] >5
47:0	FIXED_COUNTER Top-down frontend-bound counter.	
63:48	Reserved.	
Register Address: 1997H, 6551	IA32_PMC_FX5_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 5 (R/W)		
31:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:32	Reserved.	
Register Address: 1998H, 6552	IA32_PMC_FX6_CTR	
Fixed-Function Performance Counter 6 (R/W) Top-down retiring.		If CPUID.0AH:EDX[4:0] >6
47:0	FIXED_COUNTER Top-down retiring counter.	
63:48	Reserved.	
Register Address: 199BH, 6555	IA32_PMC_FX6_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 6 (R/W)		
31:0	RELOAD_VALUE Contains the reload value to be loaded into the associated counter by Auto Counter Reload. Will be 1-extended to 48 bits.	
63:32	Reserved.	
Register Address: 1B01H, 6913	IA32_UARCH_MISC_CTL	
IA32_UARCH_MISC_CTL (R/W)		If IA32_ARCH_CAPABILITIES[12]=1
0	Data Operand Independent Timing Mode (DOITM).	If IA32_ARCH_CAPABILITIES[12]=1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	**MSR/Bit Description**	**Comment**
63:1	Reserved.	
Register Address: 4000_0000H—4000_00FFH	Reserved MSR Address Space	
All existing and future processors will not implement MSRs in this range.		
Register Address: C000_0080H	IA32_EFER	
Extended Feature Enables		If (CPUID.80000001H:EDX.[20] || CPUID.80000001H:EDX.[29])
0	SYSCALL Enable: IA32_EFER.SCE (R/W) Enables SYSCALL/SYSRET instructions in 64-bit mode.	
7:1	Reserved.	
8	IA-32e Mode Enable: IA32_EFER.LME (R/W) Enables IA-32e mode operation.	
9	Reserved.	
10	IA-32e Mode Active: IA32_EFER.LMA (R) Indicates IA-32e mode is active when set.	
11	Execute Disable Bit Enable: IA32_EFER.NXE (R/W)	
63:12	Reserved.	
Register Address: C000_0081H	IA32_STAR	
System Call Target Address (R/W)		If CPUID.80000001:EDX.[29] = 1
Register Address: C000_0082H	IA32_LSTAR	
IA-32e Mode System Call Target Address (R/W) Target RIP for the called procedure when SYSCALL is executed in 64-bit mode.		If CPUID.80000001:EDX.[29] = 1
Register Address: C000_0083H	IA32_CSTAR	
IA-32e Mode System Call Target Address (R/W) Not used, as the SYSCALL instruction is not recognized in compatibility mode.		If CPUID.80000001:EDX.[29] = 1
Register Address: C000_0084H	IA32_FMASK	
System Call Flag Mask (R/W)		If CPUID.80000001:EDX.[29] = 1
Register Address: C000_0100H	IA32_FS_BASE	
Map of BASE Address of FS (R/W)		If CPUID.80000001:EDX.[29] = 1
Register Address: C000_0101H	IA32_GS_BASE	
Map of BASE Address of GS (R/W)		If CPUID.80000001:EDX.[29] = 1
Register Address: C000_0102H	IA32_KERNEL_GS_BASE	
Swap Target of BASE Address of GS (R/W)		If CPUID.80000001:EDX.[29] = 1
Register Address: C000_0103H	IA32_TSC_AUX	
Auxiliary TSC (R/W)		If CPUID.80000001H: EDX[27] = 1 or CPUID.(EAX=7,ECX=0):ECX[bit 22] = 1
31:0	AUX: Auxiliary signature of TSC.	
63:32	Reserved.	

NOTES:

1. Some older processors may have supported this MSR as model-specific and do not enumerate it with CPUID.
2. In processors based on Intel NetBurst® microarchitecture, MSR addresses 180H-197H are supported, software must treat them as model-specific. Starting with Intel Core Duo processors, MSR addresses 180H-185H, 188H-197H are reserved.
3. The *_ADDR MSRs may or may not be present; this depends on flag settings in IA32_MC*i*_STATUS. See Section 17.3.2.3 and Section 17.3.2.4 for more information.
4. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].
5. Further details on Key Locker and usage of this MSR can be found here:

   https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

2.2 MSRS IN THE INTEL® CORE™ 2 PROCESSOR FAMILY

Table 2-3 lists model-specific registers (MSRs) for the Intel Core 2 processor family and for Intel Xeon processors based on Intel Core microarchitecture, architectural MSR addresses are also included in Table 2-3. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_0FH, see Table 2-1.

MSRs listed in Table 2-2 and Table 2-3 are also supported by processors based on the Enhanced Intel Core microarchitecture. Processors based on the Enhanced Intel Core microarchitecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_17H.

The column "Shared/Unique" applies to multi-core processors based on Intel Core microarchitecture. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 0H, 0	IA32_P5_MC_ADDR	
See Section 2.23, "MSRs in Pentium Processors."		Unique
Register Address: 1H, 1	IA32_P5_MC_TYPE	
See Section 2.23, "MSRs in Pentium Processors."		Unique
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.		Unique
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER	
See Section 19.17, "Time-Stamp Counter," and Table 2-2.		Unique
Register Address: 17H, 23	IA32_PLATFORM_ID	
Platform ID (R) See Table 2-2.		Shared
Register Address: 17H, 23	MSR_PLATFORM_ID	
Model Specific Platform ID (R)		Shared
7:0	Reserved.	
12:8	Maximum Qualified Ratio (R) The maximum allowed bus ratio.	
49:13	Reserved.	
52:50	See Table 2-2.	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
63:53	Reserved.	
Register Address: 1BH, 27	IA32_APIC_BASE	
See Section 12.4.4, "Local APIC Status and Location," and Table 2-2.		Unique
Register Address: 2AH, 42	MSR_EBL_CR_POWERON	
Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.		Shared
0	Reserved.	
1	Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processors implement R/W.	
2	Response Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.	
3	MCERR# Drive Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processors implement R/W.	
4	Address Parity Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processors implement R/W.	
5	Reserved.	
6	Reserved.	
7	BINIT# Driver Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processors implement R/W.	
8	Output Tri-state Enabled (R/O) 1 = Enabled; 0 = Disabled.	
9	Execute BIST (R/O) 1 = Enabled; 0 = Disabled.	
10	MCERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled.	
11	Intel TXT Capable Chipset. (R/O) 1 = Present; 0 = Not Present.	
12	BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled.	
13	Reserved.	
14	1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes.	
15	Reserved.	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
17:16	APIC Cluster ID (R/O)	
18	N/2 Non-Integer Bus Ratio (R/O) 0 = Integer ratio; 1 = Non-integer ratio.	
19	Reserved.	
21: 20	Symmetric Arbitration ID (R/O)	
26:22	Integer Bus Frequency Ratio (R/O)	
Register Address: 3AH, 58	MSR_FEATURE_CONTROL	
Control Features in Intel 64 Processor (R/W) See Table 2-2.		Unique
3	SMRR Enable (R/WL) When this bit is set and the lock bit is set, this makes the SMRR_PHYS_BASE and SMRR_PHYS_MASK registers read visible and writeable while in SMM.	Unique
Register Address: 40H, 64	MSR_LASTBRANCH_0_FROM_IP	
Last Branch Record 0 From IP (R/W) One of four pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.5.		Unique
Register Address: 41H, 65	MSR_LASTBRANCH_1_FROM_IP	
Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 42H, 66	MSR_LASTBRANCH_2_FROM_IP	
Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 43H, 67	MSR_LASTBRANCH_3_FROM_IP	
Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 60H, 96	MSR_LASTBRANCH_0_TO_IP	
Last Branch Record 0 To IP (R/W) One of four pairs of last branch record registers on the last branch record stack. This To_IP part of the stack contains pointers to the destination instruction.		Unique
Register Address: 61H, 97	MSR_LASTBRANCH_1_TO_IP	
Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 62H, 98	MSR_LASTBRANCH_2_TO_IP	
Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 63H, 99	MSR_LASTBRANCH_3_TO_IP	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG	
BIOS Update Trigger Register (W) See Table 2-2.		Unique
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Unique
Register Address: A0H, 160	MSR_SMRR_PHYSBASE	
System Management Mode Base Address register (WO in SMM) Model-specific implementation of SMRR-like interface, read visible and write only in SMM.		Unique
11:0	Reserved.	
31:12	PhysBase: SMRR physical Base Address.	
63:32	Reserved.	
Register Address: A1H, 161	MSR_SMRR_PHYSMASK	
System Management Mode Physical Address Mask register (WO in SMM) Model-specific implementation of SMRR-like interface, read visible and write only in SMM.		Unique
10:0	Reserved.	
11	Valid: Physical address base and range mask are valid.	
31:12	PhysMask: SMRR physical address range mask.	
63:32	Reserved.	
Register Address: C1H, 193	IA32_PMC0	
Performance Counter Register See Table 2-2.		Unique
Register Address: C2H, 194	IA32_PMC1	
Performance Counter Register See Table 2-2.		Unique
Register Address: CDH, 205	MSR_FSB_FREQ	
Scaleable Bus Speed (R/O) This field indicates the intended scalable bus clock speed for processors based on Intel Core microarchitecture.		Shared
2:0	▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) ▪ 010B: 200 MHz (FSB 800) ▪ 000B: 267 MHz (FSB 1067) ▪ 100B: 333 MHz (FSB 1333)	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	Bit Description	Shared/ Unique
	133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B. 266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B.	
63:3	Reserved.	
Register Address: CDH, 205	MSR_FSB_FREQ	
Scaleable Bus Speed (R/O) This field indicates the intended scalable bus clock speed for processors based on Enhanced Intel Core microarchitecture.		Shared
2:0	▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) ▪ 010B: 200 MHz (FSB 800) ▪ 000B: 267 MHz (FSB 1067) ▪ 100B: 333 MHz (FSB 1333) ▪ 110B: 400 MHz (FSB 1600) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B. 266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 110B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 111B.	
63:3	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Unique
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Unique
Register Address: FEH, 254	IA32_MTRRCAP	
See Table 2-2.		Unique
11	SMRR Capability Using MSR 0A0H and 0A1H (R)	Unique
Register Address: 174H, 372	IA32_SYSENTER_CS	
See Table 2-2.		Unique
Register Address: 175H, 373	IA32_SYSENTER_ESP	
See Table 2-2.		Unique

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 176H, 374	IA32_SYSENTER_EIP	
See Table 2-2.		Unique
Register Address: 179H, 377	IA32_MCG_CAP	
See Table 2-2.		Unique
Register Address: 17AH, 378	IA32_MCG_STATUS	
Global Machine Check Status		Unique
0	RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.	
1	EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.	
2	MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.	
63:3	Reserved.	
Register Address: 186H, 390	IA32_PERFEVTSEL0	
See Table 2-2.		Unique
Register Address: 187H, 391	IA32_PERFEVTSEL1	
See Table 2-2.		Unique
Register Address: 198H, 408	IA32_PERF_STATUS	
See Table 2-2.		Shared
Register Address: 198H, 408	MSR_PERF_STATUS	
Current performance status. See Section 16.1.1, “Software Interface For Initiating Performance State Transitions.”		Shared
15:0	Current Performance State Value	
30:16	Reserved.	
31	XE Operation (R/O). If set, XE operation is enabled. Default is cleared.	
39:32	Reserved.	
44:40	Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.	
45	Reserved.	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
46	Non-Integer Bus Ratio (R/O) Indicates non-integer bus ratio is enabled. Applies processors based on Enhanced Intel Core microarchitecture.	
63:47	Reserved.	
Register Address: 199H, 409	IA32_PERF_CTL	
See Table 2-2.		Unique
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.		Unique
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) See Table 2-2.		Unique
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Unique
Register Address: 19DH, 413	MSR_THERM2_CTL	
Thermal Monitor 2 Control		Unique
15:0	Reserved.	
16	TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle). 1 = Thermal Monitor 2 (thermally-initiated frequency transitions). If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.	
63:16	Reserved.	
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		
0	Fast-Strings Enable See Table 2-2.	
2:1	Reserved.	
3	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2.	Unique
6:4	Reserved.	
7	Performance Monitoring Available (R) See Table 2-2.	Shared

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
8	Reserved.	
9	Hardware Prefetcher Disable (R/W) When set, disables the hardware prefetcher operation on streams of data. When clear (default), enables the prefetch queue. Disabling of the hardware prefetcher may impact processor performance.	
10	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor. 0 = Indicates compatible FERR# signaling behavior. This bit must be set to 1 to support XAPIC interrupt model usage.	Shared
11	Branch Trace Storage Unavailable (R/O) See Table 2-2.	Shared
12	Processor Event Based Sampling Unavailable (R/O) See Table 2-2.	Shared
13	TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.	Shared
15:14	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.	Shared
18	ENABLE MONITOR FSM (R/W) See Table 2-2.	Shared
19	Adjacent Cache Line Prefetch Disable (R/W) When set to 1, the processor fetches the cache line that contains data currently required by the processor. When set to 0, the processor fetches cache lines that comprise a cache line pair (128 bytes). Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.	Shared

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
20	Enhanced Intel SpeedStep Technology Select Lock (R/WO) When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit). ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.	Shared
21	Reserved.	
22	Limit CPUID Maxval (R/W) See Table 2-2.	Shared
23	xTPR Message Disable (R/W) See Table 2-2.	Shared
33:24	Reserved.	
34	XD Bit Disable (R/W) When set to 1, the Execute Disable Bit feature (XD Bit) is disabled and the XD Bit extended feature flag will be clear (CPUID.80000001H: EDX[20]=0). When set to a 0 (default), the Execute Disable Bit feature (if available) allows the OS to enable PAE paging and take advantage of data only pages. BIOS must not alter the contents of this bit location if XD bit is not supported. Writing this bit to 1 when the XD Bit extended feature flag is set to 0 may generate a #GP exception.	Unique
36:35	Reserved.	
37	DCU Prefetcher Disable (R/W) When set to 1, the DCU L1 data cache prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The DCU prefetcher is an L1 data cache prefetcher. When the DCU prefetcher detects multiple loads from the same line done within a time limit, the DCU prefetcher assumes the next line will be required. The next line is prefetched in to the L1 data cache from memory or L2.	Unique
38	IDA Disable (R/W) When set to 1 on processors that support IDA, the Intel Dynamic Acceleration feature (IDA) is disabled and the IDA_Enable feature flag will be cleared (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of IDA is enabled. Note: The power-on default value is used by BIOS to detect hardware support of IDA. If the power-on default value is 1, IDA is available in the processor. If the power-on default value is 0, IDA is not available.	Shared

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
39	IP Prefetcher Disable (R/W) When set to 1, the IP prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The IP prefetcher is an L1 data cache prefetcher. The IP prefetcher looks for sequential load history to determine whether to prefetch the next expected data into the L1 cache from memory or L2.	Unique
63:40	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).		Unique
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W) See Table 2-2.		Unique
Register Address: 1DDH, 477	MSR_LER_FROM_LIP	
Last Exception Record From Linear IP (R/W) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Unique
Register Address: 1DEH, 478	MSR_LER_TO_LIP	
Last Exception Record To Linear IP (R/W) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Unique
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0	
See Table 2-2.		Unique
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0	
See Table 2-2.		Unique
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1	
See Table 2-2.		Unique
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1	
See Table 2-2.		Unique
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2	
See Table 2-2.		Unique
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2	
See Table 2-2.		Unique
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3	
See Table 2-2.		Unique
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3	
See Table 2-2.		Unique

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4	
See Table 2-2.		Unique
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4	
See Table 2-2.		Unique
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5	
See Table 2-2.		Unique
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5	
See Table 2-2.		Unique
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6	
See Table 2-2.		Unique
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6	
See Table 2-2.		Unique
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7	
See Table 2-2.		Unique
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7	
See Table 2-2.		Unique
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000	
See Table 2-2.		Unique
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000	
See Table 2-2.		Unique
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000	
See Table 2-2.		Unique
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000	
See Table 2-2.		Unique
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000	
See Table 2-2.		Unique
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000	
See Table 2-2.		Unique
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000	
See Table 2-2.		Unique
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000	
See Table 2-2.		Unique
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000	
See Table 2-2.		Unique
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
See Table 2-2.		Unique
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000	
See Table 2-2.		Unique
Register Address: 277H, 631	IA32_PAT	
See Table 2-2.		Unique
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE	
Default Memory Types (R/W) See Table 2-2.		Unique
Register Address: 309H, 777	IA32_FIXED_CTR0	
Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.		Unique
Register Address: 30AH, 778	IA32_FIXED_CTR1	
Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.		Unique
Register Address: 30BH, 779	IA32_FIXED_CTR2	
Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.		Unique
Register Address: 345H, 837	IA32_PERF_CAPABILITIES	
See Table 2-2. See Section 19.4.1, "IA32_DEBUGCTL MSR."		Unique
Register Address: 345H, 837	MSR_PERF_CAPABILITIES	
R/O. This applies to processors that do not support architectural PerfMon version 2.		Unique
5:0	LBR Format. See Table 2-2.	
6	PEBS Record Format.	
7	PEBSSaveArchRegs. See Table 2-2.	
63:8	Reserved.	
Register Address: 38DH, 909	IA32_FIXED_CTR_CTRL	
Fixed-Function-Counter Control Register (R/W) See Table 2-2.		Unique
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Unique
Register Address: 38EH, 910	MSR_PERF_GLOBAL_STATUS	
See Section 21.6.2.2, "Global Counter Control Facilities."		Unique
Register Address: 38FH, 911	IA32_PERF_GLOBAL_CTRL	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Unique
Register Address: 38FH, 911	MSR_PERF_GLOBAL_CTRL	
See Section 21.6.2.2, "Global Counter Control Facilities."		Unique

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 390H, 912	IA32_PERF_GLOBAL_OVF_CTRL	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Unique
Register Address: 390H, 912	MSR_PERF_GLOBAL_OVF_CTRL	
See Section 21.6.2.2, "Global Counter Control Facilities."		Unique
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Table 2-2. See Section 21.6.2.4, "Processor Event Based Sampling (PEBS)."		Unique
0	Enable PEBS on IA32_PMC0. (R/W)	
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Unique
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Unique
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Unique
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Unique
Register Address: 406H, 1030	IA32_MC1_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 408H, 1032	IA32_MC2_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Unique
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Unique
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 40CH, 1036	IA32_MC4_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Unique
Register Address: 40DH, 1037	IA32_MC4_STATUS	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Unique
Register Address: 40EH, 1038	IA32_MC4_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 410H, 1040	IA32_MC3_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		
Register Address: 411H, 1041	IA32_MC3_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		
Register Address: 412H, 1042	IA32_MC3_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 413H, 1043	IA32_MC3_MISC	
Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.		Unique
Register Address: 414H, 1044	IA32_MC5_CTL	
Machine Check Error Reporting Register: Controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).		Unique
Register Address: 415H, 1045	IA32_MC5_STATUS	
Machine Check Error Reporting Register: Contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.		Unique
Register Address: 416H, 1046	IA32_MC5_ADDR	
Machine Check Error Reporting Register: Contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MCi_STATUS register is set.		Unique
Register Address: 417H, 1047	IA32_MC5_MISC	
Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.		Unique
Register Address: 419H, 1045	IA32_MC6_STATUS	
Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 25.		Unique
Register Address: 480H, 1152	IA32_VMX_BASIC	
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."		Unique
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."		Unique
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Unique
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	
Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."		Unique
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."		Unique
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."		Unique
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."		Unique
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."		Unique
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."		Unique
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."		Unique
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	
Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."		Unique
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2	
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Unique
Register Address: 600H, 1536	IA32_DS_AREA	
DS Save Area (R/W) See Table 2-2. See Section 21.6.3.4, "Debug Store (DS) Mechanism."		Unique
Register Address: 107CCH, 67532	MSR_EMON_L3_CTR_CTL0	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
GBUSQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107CDH, 67533	MSR_EMON_L3_CTR_CTL1	
GBUSQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107CEH, 67534	MSR_EMON_L3_CTR_CTL2	
GSNPQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107CFH, 67535	MSR_EMON_L3_CTR_CTL3	
GSNPQ Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107D0H, 67536	MSR_EMON_L3_CTR_CTL4	
FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107D1H, 67537	MSR_EMON_L3_CTR_CTL5	
FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107D2H, 67538	MSR_EMON_L3_CTR_CTL6	
FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107D3H, 67539	MSR_EMON_L3_CTR_CTL7	
FSB Event Control/Counter Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: 107D8H, 67544	MSR_EMON_L3_GL_CTL	
L3/FSB Common Control Register (R/W) Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 19.2.2.		Unique
Register Address: C000_0080H	IA32_EFER	
Extended Feature Enables See Table 2-2.		Unique
Register Address: C000_0081H	IA32_STAR	
System Call Target Address (R/W) See Table 2-2.		Unique
Register Address: C000_0082H	IA32_LSTAR	
IA-32e Mode System Call Target Address (R/W) See Table 2-2.		Unique
Register Address: C000_0084H	IA32_FMASK	

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
System Call Flag Mask (R/W) See Table 2-2.		Unique
Register Address: C000_0100H	IA32_FS_BASE	
Map of BASE Address of FS (R/W) See Table 2-2.		Unique
Register Address: C000_0101H	IA32_GS_BASE	
Map of BASE Address of GS (R/W) See Table 2-2.		Unique
Register Address: C000_0102H	IA32_KERNEL_GS_BASE	
Swap Target of BASE Address of GS (R/W) See Table 2-2.		Unique

2.3 MSRS IN THE 45 NM AND 32 NM INTEL ATOM® PROCESSOR FAMILY

Table 2-4 lists model-specific registers (MSRs) for 45 nm and 32 nm Intel Atom processors, architectural MSR addresses are also included in Table 2-4. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_1CH, 06_26H, 06_27H, 06_35H, or 06_36H; see Table 2-1.

The column "Shared/Unique" applies to logical processors sharing the same core in processors based on the Intel Atom microarchitecture. "Unique" means each logical processor has a separate MSR, or a bit field in an MSR governs only a logical processor. "Shared" means the MSR or the bit field in an MSR address governs the operation of both logical processors in the same core.

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 0H, 0	IA32_P5_MC_ADDR	
See Section 2.23, "MSRs in Pentium Processors."		Shared
Register Address: 1H, 1	IA32_P5_MC_TYPE	
See Section 2.23, "MSRs in Pentium Processors."		Shared
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.		Unique
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER	
See Section 19.17, "Time-Stamp Counter," and see Table 2-2.		Unique
Register Address: 17H, 23	IA32_PLATFORM_ID	
Platform ID (R) See Table 2-2.		Shared
Register Address: 17H, 23	MSR_PLATFORM_ID	

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Model Specific Platform ID (R)		Shared
7:0	Reserved.	
12:8	Maximum Qualified Ratio (R) The maximum allowed bus ratio.	
63:13	Reserved.	
Register Address: 1BH, 27	IA32_APIC_BASE	
See Section 12.4.4, "Local APIC Status and Location," and Table 2-2.		Unique
Register Address: 2AH, 42	MSR_EBL_CR_POWERON	
Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.		Shared
0	Reserved.	
1	Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Always 0.	
2	Response Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Always 0.	
3	AERR# Drive Enable (R/W) 1 = Enabled; 0 = Disabled. Always 0.	
4	BERR# Enable for initiator bus requests (R/W) 1 = Enabled; 0 = Disabled. Always 0.	
5	Reserved.	
6	Reserved.	
7	BINIT# Driver Enable (R/W) 1 = Enabled; 0 = Disabled. Always 0.	
8	Reserved.	
9	Execute BIST (R/O) 1 = Enabled; 0 = Disabled.	
10	AERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled. Always 0.	
11	Reserved.	
12	BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled. Always 0.	
13	Reserved.	

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
14	1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes.	
15	Reserved.	
17:16	APIC Cluster ID (R/O) Always 00B.	
19: 18	Reserved.	
21: 20	Symmetric Arbitration ID (R/O) Always 00B.	
26:22	Integer Bus Frequency Ratio (R/O)	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64Processor (R/W) See Table 2-2.		Unique
Register Address: 40H, 64	MSR_LASTBRANCH_0_FROM_IP	
Last Branch Record 0 From IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.5.		Unique
Register Address: 41H, 65	MSR_LASTBRANCH_1_FROM_IP	
Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 42H, 66	MSR_LASTBRANCH_2_FROM_IP	
Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 43H, 67	MSR_LASTBRANCH_3_FROM_IP	
Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 44H, 68	MSR_LASTBRANCH_4_FROM_IP	
Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 45H, 69	MSR_LASTBRANCH_5_FROM_IP	
Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 46H, 70	MSR_LASTBRANCH_6_FROM_IP	
Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique
Register Address: 47H, 71	MSR_LASTBRANCH_7_FROM_IP	
Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Unique

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 60H, 96	MSR_LASTBRANCH_0_TO_IP	
Last Branch Record 0 To IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.		Unique
Register Address: 61H, 97	MSR_LASTBRANCH_1_TO_IP	
Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 62H, 98	MSR_LASTBRANCH_2_TO_IP	
Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 63H, 99	MSR_LASTBRANCH_3_TO_IP	
Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 64H, 100	MSR_LASTBRANCH_4_TO_IP	
Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 65H, 101	MSR_LASTBRANCH_5_TO_IP	
Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 66H, 102	MSR_LASTBRANCH_6_TO_IP	
Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 67H, 103	MSR_LASTBRANCH_7_TO_IP	
Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Unique
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG	
BIOS Update Trigger Register (W) See Table 2-2.		Shared
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Unique
Register Address: C1H, 193	IA32_PMC0	
Performance counter register See Table 2-2.		Unique
Register Address: C2H, 194	IA32_PMC1	
Performance Counter Register See Table 2-2.		Unique
Register Address: CDH, 205	MSR_FSB_FREQ	

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Scaleable Bus Speed (R/O) This field indicates the intended scalable bus clock speed for processors based on Intel Atom microarchitecture.		Shared
2:0	▪ 111B: 083 MHz (FSB 333) ▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.	
63:3	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Unique
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Unique
Register Address: FEH, 254	IA32_MTRRCAP	
Memory Type Range Register (R) See Table 2-2.		Shared
Register Address: 11EH, 281	MSR_BBL_CR_CTL3	
Control Register 3 Used to configure the L2 Cache.		Shared
0	L2 Hardware Enabled (R/O) 1 = Indicates the L2 is hardware-enabled. 0 = Indicates the L2 is hardware-disabled.	
7:1	Reserved.	
8	L2 Enabled (R/W) 1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set, the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.	
22:9	Reserved.	
23	L2 Not Present (R/O) 0 = L2 Present. 1 = L2 Not Present.	
63:24	Reserved.	
Register Address: 174H, 372	IA32_SYSENTER_CS	
See Table 2-2.		Unique
Register Address: 175H, 373	IA32_SYSENTER_ESP	
See Table 2-2.		Unique

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 176H, 374	IA32_SYSENTER_EIP	
See Table 2-2.		Unique
Register Address: 179H, 377	IA32_MCG_CAP	
See Table 2-2.		Unique
Register Address: 17AH, 378	IA32_MCG_STATUS	
Global Machine Check Status		Unique
0	RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.	
1	EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.	
2	MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.	
63:3	Reserved.	
Register Address: 186H, 390	IA32_PERFEVTSEL0	
See Table 2-2.		Unique
Register Address: 187H, 391	IA32_PERFEVTSEL1	
See Table 2-2.		Unique
Register Address: 198H, 408	IA32_PERF_STATUS	
See Table 2-2.		Shared
Register Address: 198H, 408	MSR_PERF_STATUS	
Performance Status		Shared
15:0	Current Performance State Value.	
39:16	Reserved.	
44:40	Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.	
63:45	Reserved.	
Register Address: 199H, 409	IA32_PERF_CTL	
See Table 2-2.		Unique
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.		Unique

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) See Table 2-2.		Unique
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Unique
Register Address: 19DH, 413	MSR_THERM2_CTL	
Thermal Monitor 2 Control		Shared
15:0	Reserved.	
16	TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle). 1 = Thermal Monitor 2 (thermally-initiated frequency transitions). If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.	
63:17	Reserved.	
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		Unique
0	Fast-Strings Enable See Table 2-2.	
2:1	Reserved.	
3	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 0.	Unique
6:4	Reserved.	
7	Performance Monitoring Available (R) See Table 2-2.	Shared
8	Reserved.	
9	Reserved.	
10	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor. 0 = Indicates compatible FERR# signaling behavior. This bit must be set to 1 to support XAPIC interrupt model usage.	Shared
11	Branch Trace Storage Unavailable (R/O) See Table 2-2.	Shared
12	Processor Event Based Sampling Unavailable (R/O) See Table 2-2.	Shared

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
13	TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is cleared (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.	Shared
15:14	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.	Shared
18	ENABLE MONITOR FSM (R/W) See Table 2-2.	Shared
19	Reserved.	
20	Enhanced Intel SpeedStep Technology Select Lock (R/WO) When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit). ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.	Shared
21	Reserved.	
22	Limit CPUID Maxval (R/W) See Table 2-2.	Unique
23	xTPR Message Disable (R/W) See Table 2-2.	Shared
33:24	Reserved.	
34	XD Bit Disable (R/W) See Table 2-3.	Unique
63:35	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).		Unique
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W) See Table 2-2.		Unique

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 1DDH, 477	MSR_LER_FROM_LIP	
Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Unique
Register Address: 1DEH, 478	MSR_LER_TO_LIP	
Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Unique
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0	
See Table 2-2.		Shared
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0	
See Table 2-2.		Shared
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1	
See Table 2-2.		Shared
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1	
See Table 2-2.		Shared
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2	
See Table 2-2.		Shared
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2	
See Table 2-2.		Shared
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3	
See Table 2-2.		Shared
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3	
See Table 2-2.		Shared
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4	
See Table 2-2.		Shared
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4	
See Table 2-2.		Shared
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5	
See Table 2-2.		Shared
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5	
See Table 2-2.		Shared
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6	
See Table 2-2.		Shared
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6	
See Table 2-2.		Shared
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7	
See Table 2-2.		Shared

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7	
See Table 2-2.		Shared
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000	
See Table 2-2.		Shared
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000	
See Table 2-2.		Shared
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000	
See Table 2-2.		Shared
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000	
See Table 2-2.		Shared
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000	
See Table 2-2.		Shared
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000	
See Table 2-2.		Shared
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000	
See Table 2-2.		Shared
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000	
See Table 2-2.		Shared
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000	
See Table 2-2.		Shared
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000	
See Table 2-2.		Shared
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000	
See Table 2-2.		Shared
Register Address: 277H, 631	IA32_PAT	
See Table 2-2.		Unique
Register Address: 309H, 777	IA32_FIXED_CTR0	
Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.		Unique
Register Address: 30AH, 778	IA32_FIXED_CTR1	
Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.		Unique
Register Address: 30BH, 779	IA32_FIXED_CTR2	
Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.		Unique
Register Address: 345H, 837	IA32_PERF_CAPABILITIES	
See Table 2-2. See Section 19.4.1, "IA32_DEBUGCTL MSR."		Shared

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 38DH, 909	IA32_FIXED_CTR_CTRL	
Fixed-Function-Counter Control Register (R/W) See Table 2-2.		Unique
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2. See Section 21.6.2.2, “Global Counter Control Facilities.”		Unique
Register Address: 38FH, 911	IA32_PERF_GLOBAL_CTRL	
See Table 2-2. See Section 21.6.2.2, “Global Counter Control Facilities.”		Unique
Register Address: 390H, 912	IA32_PERF_GLOBAL_OVF_CTRL	
See Table 2-2. See Section 21.6.2.2, “Global Counter Control Facilities.”		Unique
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Table 2-2. See Section 21.6.2.4, “Processor Event Based Sampling (PEBS).”		Unique
0	Enable PEBS on IA32_PMC0 (R/W)	
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Shared
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS.”		Shared
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Shared
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Shared
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS.”		Shared
Register Address: 408H, 1032	IA32_MC2_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Shared
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS.”		Shared
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Shared
Register Address: 40CH, 1036	IA32_MC3_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Shared
Register Address: 40DH, 1037	IA32_MC3_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS.”		Shared

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 40EH, 1038	IA32_MC3_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Shared
Register Address: 410H, 1040	IA32_MC4_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Shared
Register Address: 411H, 1041	IA32_MC4_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Shared
Register Address: 412H, 1042	IA32_MC4_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Shared
Register Address: 480H, 1152	IA32_VMX_BASIC	
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."		Unique
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."		Unique
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Unique
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	
Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."		Unique
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."		Unique
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."		Unique
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."		Unique
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."		Unique
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."		Unique
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."		Unique
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	
Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."		Unique
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2	
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Unique
Register Address: 600H, 1536	IA32_DS_AREA	
DS Save Area (R/W) See Table 2-2. See Section 21.6.3.4, "Debug Store (DS) Mechanism."		Unique
Register Address: C000_0080H	IA32_EFER	
Extended Feature Enables See Table 2-2.		Unique
Register Address: C000_0081H	IA32_STAR	
System Call Target Address (R/W) See Table 2-2.		Unique
Register Address: C000_0082H	IA32_LSTAR	
IA-32e Mode System Call Target Address (R/W) See Table 2-2.		Unique
Register Address: C000_0084H	IA32_FMASK	
System Call Flag Mask (R/W) See Table 2-2.		Unique
Register Address: C000_0100H	IA32_FS_BASE	
Map of BASE Address of FS (R/W) See Table 2-2.		Unique
Register Address: C000_0101H	IA32_GS_BASE	
Map of BASE Address of GS (R/W) See Table 2-2.		Unique
Register Address: C000_0102H	IA32_KERNEL_GS_BASE	
Swap Target of BASE Address of GS (R/W) See Table 2-2.		Unique

Table 2-5 lists model-specific registers (MSRs) that are specific to Intel Atom® processor with a CPUID Signature DisplayFamily_DisplayModel value of 06_27H.

Table 2-5. MSRs Supported by Intel Atom® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_27H

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3F8H, 1016	MSR_PKG_C2_RESIDENCY	
Package C2 Residency Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C2 Residency Counter (R/O) Time that this package is in processor-specific C2 states since last reset. Counts at 1 Mhz frequency.	Package
Register Address: 3F9H, 1017	MSR_PKG_C4_RESIDENCY	
Package C4 Residency Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C4 Residency Counter. (R/O) Time that this package is in processor-specific C4 states since last reset. Counts at 1 Mhz frequency.	Package
Register Address: 3FAH, 1018	MSR_PKG_C6_RESIDENCY	
Package C6 Residency Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C6 Residency Counter. (R/O) Time that this package is in processor-specific C6 states since last reset. Counts at 1 Mhz frequency.	Package

2.4 MSRS IN INTEL PROCESSORS BASED ON SILVERMONT MICROARCHITECTURE

Table 2-6 lists model-specific registers (MSRs) common to Intel processors based on the Silvermont microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_37H, 06_4AH, 06_4DH, 06_5AH, or 06_5DH; see Table 2-1. The MSRs listed in Table 2-6 are also common to processors based on the Airmont microarchitecture and newer microarchitectures for next generation Intel Atom processors.

Table 2-7 lists MSRs common to processors based on the Silvermont and Airmont microarchitectures, but not newer microarchitectures.

Table 2-8, Table 2-9, and Table 2-10 lists MSRs that are model-specific across processors based on the Silvermont microarchitecture.

In the Silvermont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Silvermont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 0H, 0	IA32_P5_MC_ADDR	
See Section 2.23, "MSRs in Pentium Processors."		Module
Register Address: 1H, 1	IA32_P5_MC_TYPE	
See Section 2.23, "MSRs in Pentium Processors."		Module
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.		Core
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER	
See Section 19.17, "Time-Stamp Counter," and Table 2-2.		Core
Register Address: 1BH, 27	IA32_APIC_BASE	
See Section 12.4.4, "Local APIC Status and Location," and Table 2-2.		Core
Register Address: 2AH, 42	MSR_EBL_CR_POWERON	
Processor Hard Power-On Configuration (R/W) Writes ignored.		Module
63:0	Reserved.	
Register Address: 34H, 52	MSR_SMI_COUNT	
SMI Counter (R/O)		Core
31:0	SMI Count (R/O) Running count of SMI events since last RESET.	
63:32	Reserved.	
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG	
BIOS Update Trigger Register (W) See Table 2-2.		Core
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Core
Register Address: C1H, 193	IA32_PMC0	
Performance Counter Register See Table 2-2.		Core
Register Address: C2H, 194	IA32_PMC1	
Performance Counter Register See Table 2-2.		Core
Register Address: E4H, 228	MSR_PMG_IO_CAPTURE_BASE	
Power Management IO Redirection in C-state (R/W) See http://biosbits.org.		Module

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
15:0	LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.	
18:16	C-state Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 100b - C4 is the max C-State to include 110b - C6 is the max C-State to include 111b - C7 is the max C-State to include	
63:19	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Core
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Core
Register Address: FEH, 254	IA32_MTRRCAP	
Memory Type Range Register (R) See Table 2-2.		Core
Register Address: 13CH, 316	MSR_FEATURE_CONFIG	
AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.		Core
1:0	AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction sets availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note: AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.	
63:2	Reserved.	
Register Address: 174H, 372	IA32_SYSENTER_CS	
See Table 2-2.		Core
Register Address: 175H, 373	IA32_SYSENTER_ESP	
See Table 2-2.		Core
Register Address: 176H, 374	IA32_SYSENTER_EIP	
See Table 2-2.		Core
Register Address: 179H, 377	IA32_MCG_CAP	

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2.		Core
Register Address: 17AH, 378	IA32_MCG_STATUS	
Global Machine Check Status		Core
0	RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.	
1	EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.	
2	MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.	
63:3	Reserved.	
Register Address: 186H, 390	IA32_PERFEVTSEL0	
See Table 2-2.		Core
7:0	Event Select	
15:8	UMask	
16	USR	
17	OS	
18	Edge	
19	PC	
20	INT	
21	Reserved.	
22	EN	
23	INV	
31:24	CMASK	
63:32	Reserved.	
Register Address: 187H, 391	IA32_PERFEVTSEL1	
See Table 2-2.		Core
Register Address: 198H, 408	IA32_PERF_STATUS	
See Table 2-2.		Module
Register Address: 199H, 409	IA32_PERF_CTL	
See Table 2-2.		Core
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.		Core
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) See Table 2-2.		Core
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Core
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target		Package
15:0	Reserved.	
23:16	Temperature Target (R) The default thermal throttling or PROCHOT# activation temperature in degrees C. The effective temperature for thermal throttling or PROCHOT# activation is “Temperature Target” + “Target Offset”.	
29:24	Target Offset (R/W) Specifies an offset in degrees C to adjust the throttling and PROCHOT# activation temperature from the default target specified in TEMPERATURE_TARGET (bits 23:16).	
63:30	Reserved.	
Register Address: 1A6H, 422	MSR_OFFCORE_RSP_0	
Offcore Response Event Select Register (R/W)		Module
Register Address: 1A7H, 423	MSR_OFFCORE_RSP_1	
Offcore Response Event Select Register (R/W)		Module
Register Address: 1B0H, 432	IA32_ENERGY_PERF_BIAS	
See Table 2-2.		Core
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W) See Table 2-2.		Core
Register Address: 1DDH, 477	MSR_LER_FROM_LIP	
Last Exception Record From Linear IP (R/W) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Core
Register Address: 1DEH, 478	MSR_LER_TO_LIP	
Last Exception Record To Linear IP (R/W) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Core
Register Address: 1F2H, 498	IA32_SMRR_PHYSBASE	
See Table 2-2.		Core

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1F3H, 499	IA32_SMRR_PHYSMASK	
See Table 2-2.		Core
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0	
See Table 2-2.		Core
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0	
See Table 2-2.		Core
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1	
See Table 2-2.		Core
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1	
See Table 2-2.		Core
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2	
See Table 2-2.		Core
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2	
See Table 2-2.		Core
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3	
See Table 2-2.		Core
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3	
See Table 2-2.		Core
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4	
See Table 2-2.		Core
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4	
See Table 2-2.		Core
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5	
See Table 2-2.		Core
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5	
See Table 2-2.		Core
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6	
See Table 2-2.		Core
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6	
See Table 2-2.		Core
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7	
See Table 2-2.		Core
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7	
See Table 2-2.		Core
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000	
See Table 2-2.		Core
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000	

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2.		Core
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000	
See Table 2-2.		Core
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000	
See Table 2-2.		Core
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000	
See Table 2-2.		Core
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000	
See Table 2-2.		Core
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000	
See Table 2-2.		Core
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000	
See Table 2-2.		Core
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000	
See Table 2-2.		Core
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000	
See Table 2-2.		Core
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000	
See Table 2-2.		Core
Register Address: 277H, 631	IA32_PAT	
See Table 2-2.		Core
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE	
Default Memory Types (R/W) See Table 2-2.		Core
Register Address: 309H, 777	IA32_FIXED_CTR0	
Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.		Core
Register Address: 30AH, 778	IA32_FIXED_CTR1	
Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.		Core
Register Address: 30BH, 779	IA32_FIXED_CTR2	
Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.		Core
Register Address: 345H, 837	IA32_PERF_CAPABILITIES	
See Table 2-2. See Section 19.4.1, "IA32_DEBUGCTL MSR."		Core
Register Address: 38DH, 909	IA32_FIXED_CTR_CTRL	
Fixed-Function-Counter Control Register (R/W) See Table 2-2.		Core

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 38FH, 911	IA32_PERF_GLOBAL_CTRL	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Core
Register Address: 3FDH, 1021	MSR_CORE_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Counts at the TSC Frequency.	
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Module
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Module
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Module
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Module
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Module
Register Address: 408H, 1032	IA32_MC2_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Module
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Module
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Module
Register Address: 40CH, 1036	IA32_MC3_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 40DH, 1037	IA32_MC3_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 40EH, 1038	IA32_MC3_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Core
Register Address: 410H, 1040	IA32_MC4_CTL	

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 411H, 1041	IA32_MC4_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 412H, 1042	IA32_MC4_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Core
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 415H, 1045	IA32_MC5_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Package
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Package
Register Address: 480H, 1152	IA32_VMX_BASIC	
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, "Basic VMX Information."		Core
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, "VM-Execution Controls."		Core
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Core
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	
Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, "VM-Exit Controls."		Core
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, "VM-Entry Controls."		Core
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2. See Appendix A.6, "Miscellaneous Data."		Core

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."		Core
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."		Core
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."		Core
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."		Core
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	
Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2. See Appendix A.9, "VMCS Enumeration."		Core
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2	
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Core
Register Address: 48CH, 1164	IA32_VMX_EPT_VPID_ENUM	
Capability Reporting Register of EPT and VPID (R/O) See Table 2-2.		Core
Register Address: 48DH, 1165	IA32_VMX_TRUE_PINBASED_CTLS	
Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2.		Core
Register Address: 48EH, 1166	IA32_VMX_TRUE_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-based VM-Execution Flex Controls (R/O) See Table 2-2.		Core
Register Address: 48FH, 1167	IA32_VMX_TRUE_EXIT_CTLS	
Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2.		Core
Register Address: 490H, 1168	IA32_VMX_TRUE_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Flex Controls (R/O) See Table 2-2.		Core
Register Address: 491H, 1169	IA32_VMX_FMFUNC	

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Capability Reporting Register of VM-Function Controls (R/O) See Table 2-2.		Core
Register Address: 4C1H, 1217	IA32_A_PMC0	
See Table 2-2.		Core
Register Address: 4C2H, 1218	IA32_A_PMC1	
See Table 2-2.		Core
Register Address: 600H, 1536	IA32_DS_AREA	
DS Save Area (R/W) See Table 2-2 and Section 21.6.3.4, "Debug Store (DS) Mechanism."		Core
Register Address: 660H, 1632	MSR_CORE_C1_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C1 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C1 states. Counts at the TSC frequency.	
Register Address: 6E0H, 1760	IA32_TSC_DEADLINE	
TSC Target of Local APIC's TSC Deadline Mode (R/W) See Table 2-2.		Core
Register Address: C000_0080H	IA32_EFER	
Extended Feature Enables See Table 2-2.		Core
Register Address: C000_0081H	IA32_STAR	
System Call Target Address (R/W) See Table 2-2.		Core
Register Address: C000_0082H	IA32_LSTAR	
IA-32e Mode System Call Target Address (R/W) See Table 2-2.		Core
Register Address: C000_0084H	IA32_FMASK	
System Call Flag Mask (R/W) See Table 2-2.		Core
Register Address: C000_0100H	IA32_FS_BASE	
Map of BASE Address of FS (R/W) See Table 2-2.		Core
Register Address: C000_0101H	IA32_GS_BASE	
Map of BASE Address of GS (R/W) See Table 2-2.		Core
Register Address: C000_0102H	IA32_KERNEL_GS_BASE	
Swap Target of BASE Address of GS (R/W) See Table 2-2.		Core

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: C000_0103H	IA32_TSC_AUX	
AUXILIARY TSC Signature (R/W) See Table 2-2.		Core

Table 2-7 lists model-specific registers (MSRs) that are common to Intel Atom® processors based on the Silvermont and Airmont microarchitectures but not newer microarchitectures.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 17H, 23	MSR_PLATFORM_ID	
Model Specific Platform ID (R)		Module
7:0	Reserved.	
13:8	Maximum Qualified Ratio (R) The maximum allowed bus ratio.	
49:13	Reserved.	
52:50	See Table 2-2.	
63:33	Reserved.	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64Processor (R/W) See Table 2-2.		Core
0	Lock (R/WL)	
1	Reserved.	
2	Enable VMX outside SMX operation (R/WL)	
Register Address: 40H, 64	MSR_LASTBRANCH_0_FROM_IP	
Last Branch Record 0 From IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.5 and record format in Section 19.4.8.1.		Core
Register Address: 41H, 65	MSR_LASTBRANCH_1_FROM_IP	
Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 42H, 66	MSR_LASTBRANCH_2_FROM_IP	
Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 43H, 67	MSR_LASTBRANCH_3_FROM_IP	
Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 44H, 68	MSR_LASTBRANCH_4_FROM_IP	

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 45H, 69	MSR_LASTBRANCH_5_FROM_IP	
Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 46H, 70	MSR_LASTBRANCH_6_FROM_IP	
Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 47H, 71	MSR_LASTBRANCH_7_FROM_IP	
Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 60H, 96	MSR_LASTBRANCH_0_TO_IP	
Last Branch Record 0 To IP (R/W) One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.		Core
Register Address: 61H, 97	MSR_LASTBRANCH_1_TO_IP	
Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 62H, 98	MSR_LASTBRANCH_2_TO_IP	
Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 63H, 99	MSR_LASTBRANCH_3_TO_IP	
Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 64H, 100	MSR_LASTBRANCH_4_TO_IP	
Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 65H, 101	MSR_LASTBRANCH_5_TO_IP	
Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 66H, 102	MSR_LASTBRANCH_6_TO_IP	
Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 67H, 103	MSR_LASTBRANCH_7_TO_IP	
Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information: Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * Scalable Bus Frequency.	Package
63:16	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.		Module
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 100b: C4 110b: C6 111b: C7 (Silvermont only)	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.	
63:16	Reserved.	
Register Address: 11EH, 281	MSR_BBL_CR_CTL3	
Control Register 3 Used to configure the L2 Cache.		Module
0	L2 Hardware Enabled (R/O) 1 = If the L2 is hardware-enabled. 0 = Indicates if the L2 is hardware-disabled.	
7:1	Reserved.	
8	L2 Enabled (R/W) 1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.	
22:9	Reserved.	

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
23	L2 Not Present (R/O) 0 = L2 Present. 1 = L2 Not Present.	
63:24	Reserved.	
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		
0	Fast-Strings Enable See Table 2-2.	Core
2:1	Reserved.	
3	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 0.	Module
6:4	Reserved.	
7	Performance Monitoring Available (R) See Table 2-2.	Core
10:8	Reserved.	
11	Branch Trace Storage Unavailable (R/O) See Table 2-2.	Core
12	Processor Event Based Sampling Unavailable (R/O) See Table 2-2.	Core
15:13	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.	Module
18	ENABLE MONITOR FSM (R/W) See Table 2-2.	Core
21:19	Reserved.	
22	Limit CPUID Maxval (R/W) See Table 2-2.	Core
23	xTPR Message Disable (R/W) See Table 2-2.	Module
33:24	Reserved.	
34	XD Bit Disable (R/W) See Table 2-3.	Core
37:35	Reserved.	

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
38	Turbo Mode Disable (R/W) When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be cleared (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.	Module
63:39	Reserved.	
Register Address: 1C8H, 456	MSR_LBR_SELECT	
Last Branch Record Filtering Select Register (R/W) See Section 19.9.2, "Filtering of Last Branch Records."		Core
0	CPL_EQ_0	
1	CPL_NEQ_0	
2	JCC	
3	NEAR_REL_CALL	
4	NEAR_IND_CALL	
5	NEAR_RET	
6	NEAR_IND_JMP	
7	NEAR_REL_JMP	
8	FAR_BRANCH	
63:9	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Core
Register Address: 390H, 912	IA32_PERF_GLOBAL_OVF_CTRL	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Core
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Table 2-2. See Section 21.6.2.4, "Processor Event Based Sampling (PEBS)."		Core
0	Enable PEBS for precise event on IA32_PMC0 (R/W)	
Register Address: 3FAH, 1018	MSR_PKG_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C6 Residency Counter (R/O) Value since last reset that this package is in processor-specific C6 states. Counts at the TSC Frequency.	

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 664H, 1636	MSR_MC6_RESIDENCY_COUNTER	
Module C6 Residency Counter (R/0) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Module
63:0	Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.	

2.4.1 MSRs with Model-Specific Behavior in the Silvermont Microarchitecture

Table 2-8 lists MSRs that are specific to the Intel Atom® processor E3000 Series (CPUID Signature DisplayFamily_DisplayModel value of 06_37H) and Intel Atom processors (CPUID Signature DisplayFamily_DisplayModel value of 06_4AH, 06_5AH, or 06_5DH).

Table 2-8. Specific MSRs Supported by Intel Atom® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_37H, 06_4AH, 06_5AH, or 06_5DH

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: CDH, 205	MSR_FSB_FREQ	
Scaleable Bus Speed (R/O) This field indicates the intended scalable bus clock speed for processors based on Silvermont microarchitecture.		Module
2:0	▪ 100B: 080.0 MHz ▪ 000B: 083.3 MHz ▪ 001B: 100.0 MHz ▪ 010B: 133.3 MHz ▪ 011B: 116.7 MHz	
63:3	Reserved.	
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
	Unit Multipliers used in RAPL Interfaces (R/O) See Section 16.10.1, “RAPL Interfaces.”	Package
3:0	Power Units Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.	
7:4	Reserved.	
12:8	Energy Status Units Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.	
15:13	Reserved.	
19:16	Time Unit The value is 0000b, indicating time unit is in one second.	
63:20	Reserved.	
Register Address: 610H, 1552	MSR_PKG_POWER_LIMIT	
PKG RAPL Power Limit Control (R/W)		Package

Table 2-8. Specific MSRs Supported by Intel Atom® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_37H, 06_4AH, 06_5AH, or 06_5DH (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
14:0	Package Power Limit #1 (R/W) See Section 16.10.3, "Package RAPL Domain," and MSR_RAPL_POWER_UNIT in Table 2-8.	
15	Enable Power Limit #1 (R/W) See Section 16.10.3, "Package RAPL Domain."	
16	Package Clamping Limitation #1 (R/W) See Section 16.10.3, "Package RAPL Domain."	
23:17	Time Window for Power Limit #1 (R/W) In unit of second. If 0 is specified in bits [23:17], defaults to 1 second window.	
63:24	Reserved.	
Register Address: 611H, 1553	MSR_PKG_ENERGY_STATUS	
PKG Energy Status (R/O) See Section 16.10.3, "Package RAPL Domain," and MSR_RAPL_POWER_UNIT in Table 2-8.		Package
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains," and MSR_RAPL_POWER_UNIT in Table 2-8.		Package

Table 2-9 lists model-specific registers (MSRs) that are specific to the Intel Atom® processor E3000 Series (CPUID Signature DisplayFamily_DisplayModel value of 06_37H).

Table 2-9. Specific MSRs Supported by the Intel Atom® Processor E3000 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_37H

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 668H, 1640	MSR_CC6_DEMOTION_POLICY_CONFIG	
Core C6 Demotion Policy Config MSR		Package
63:0	Controls per-core C6 demotion policy. Writing a value of 0 disables core level HW demotion policy.	
Register Address: 669H, 1641	MSR_MC6_DEMOTION_POLICY_CONFIG	
Module C6 Demotion Policy Config MSR		Package
63:0	Controls module (i.e., two cores sharing the second-level cache) C6 demotion policy. Writing a value of 0 disables module level HW demotion policy.	
Register Address: 664H, 1636	MSR_MC6_RESIDENCY_COUNTER	
Module C6 Residency Counter (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Module
63:0	Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.	

Table 2-10 lists model-specific registers (MSRs) that are specific to Intel Atom® processor C2000 Series (CPUID Signature DisplayFamily_DisplayModel value of 06_4DH).

Table 2-10. Specific MSRs Supported by Intel Atom® Processor C2000 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4DH

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1A4H, 420	MSR_MISC_FEATURE_CONTROL	
Miscellaneous Feature Control (R/W)		
0	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.	Core
1	Reserved.	
2	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.	Core
63:3	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode (R/W)		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
39:32	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 core active.	Package
47:40	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 core active.	Package
55:48	Maximum Ratio Limit for 7C Maximum turbo ratio limit of 7 core active.	Package
63:56	Maximum Ratio Limit for 8C Maximum turbo ratio limit of 8 core active.	Package
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers used in RAPL Interfaces (R/O) See Section 16.10.1, "RAPL Interfaces."		Package
3:0	Power Units Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.	
7:4	Reserved.	

Table 2-10. Specific MSRs Supported by Intel Atom® Processor C2000 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4DH (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
12:8	Energy Status Units. Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.	
15:13	Reserved.	
19:16	Time Unit The value is 0000b, indicating time unit is in one second.	
63:20	Reserved.	
Register Address: 610H, 1552	MSR_PKG_POWER_LIMIT	
PKG RAPL Power Limit Control (R/W) See Section 16.10.3, "Package RAPL Domain."		Package
Register Address: 66EH, 1646	MSR_PKG_POWER_INFO	
PKG RAPL Parameter (R/0)		Package
14:0	Thermal Spec Power (R/0) The unsigned integer value is the equivalent of the thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.	
63:15	Reserved.	

2.4.2 MSRs in Intel Atom® Processors Based on Airmont Microarchitecture

Intel Atom processor X7-Z8000 and X5-Z8000 series are based on the Airmont microarchitecture. These processors support MSRs listed in Table 2-6, Table 2-7, Table 2-8, and Table 2-11. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_4CH; see Table 2-1.

Table 2-11. MSRs in Intel Atom® Processors Based on Airmont Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: CDH, 205	MSR_FSB_FREQ	
Scaleable Bus Speed (R/O) This field indicates the intended scalable bus clock speed for processors based on Airmont microarchitecture.		Module
3:0	▪ 0000B: 083.3 MHz ▪ 0001B: 100.0 MHz ▪ 0010B: 133.3 MHz ▪ 0011B: 116.7 MHz ▪ 0100B: 080.0 MHz ▪ 0101B: 093.3 MHz ▪ 0110B: 090.0 MHz ▪ 0111B: 088.9 MHz ▪ 1000B: 087.5 MHz	
63:5	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	

Table 2-11. MSRs in Intel Atom® Processors Based on Airmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.		Module
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: No limit 001b: C1 010b: C2 110b: C6 111b: C7	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.	
63:16	Reserved.	
Register Address: E4H, 228	MSR_PMG_IO_CAPTURE_BASE	
Power Management IO Redirection in C-state (R/W) See http://biosbits.org.		Module
15:0	LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.	
18:16	C-state Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include. 001b - Deep Power Down Technology is the max C-State. 010b - C7 is the max C-State to include.	
63:19	Reserved.	
Register Address: 638H, 1592	MSR_PP0_POWER_LIMIT	
PP0 RAPL Power Limit Control (R/W)		Package

Table 2-11. MSRs in Intel Atom® Processors Based on Airmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
14:0	PP0 Power Limit #1 (R/W) See Section 16.10.4, “PP0/PP1 RAPL Domains,” and MSR_RAPL_POWER_UNIT in Table 2-8.	
15	Enable Power Limit #1 (R/W) See Section 16.10.4, “PP0/PP1 RAPL Domains.”	
16	Reserved.	
23:17	Time Window for Power Limit #1 (R/W) Specifies the time duration over which the average power must remain below PP0_POWER_LIMIT #1(14:0). Supported Encodings: 0x0: 1 second time duration. 0x1: 5 second time duration (Default). 0x2: 10 second time duration. 0x3: 15 second time duration. 0x4: 20 second time duration. 0x5: 25 second time duration. 0x6: 30 second time duration. 0x7: 35 second time duration. 0x8: 40 second time duration. 0x9: 45 second time duration. 0xA: 50 second time duration. 0xB-0x7F - reserved.	
63:24	Reserved.	

2.5 MSRS IN INTEL ATOM® PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Intel Atom processors based on the Goldmont microarchitecture support MSRs listed in Table 2-6 and Table 2-12. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_5CH; see Table 2-1.

In the Goldmont microarchitecture, the scope column indicates the following: “Core” means each processor core has a separate MSR, or a bit field not shared with another processor core. “Module” means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Goldmont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. “Package” means all processor cores in the physical package share the same MSR or bit interface.

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 17H, 23	MSR_PLATFORM_ID	
Model Specific Platform ID (R)		Module
49:0	Reserved.	
52:50	See Table 2-2.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:33	Reserved.	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64 Processor (R/W) See Table 2-2.		Core
0	Lock (R/WL)	
1	Enable VMX inside SMX operation (R/WL)	
2	Enable VMX outside SMX operation (R/WL)	
14:8	SENTER local functions enables (R/WL)	
15	SENTER global functions enable (R/WL)	
18	SGX global functions enable (R/WL)	
63:19	Reserved.	
Register Address: 3BH, 59	IA32_TSC_ADJUST	
Per-Core TSC ADJUST (R/W) See Table 2-2.		Core
Register Address: C3H, 195	IA32_PMC2	
Performance Counter Register See Table 2-2.		Core
Register Address: C4H, 196	IA32_PMC3	
Performance Counter Register See Table 2-2.		Core
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * 100 MHz.	Package
27:16	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.	Package
29	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.	Package
30	Programmable TJ OFFSET (R/O) When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify a temperature offset.	Package
39:31	Reserved.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
47:40	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	Package
63:48	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.		Core
3:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: No limit 0001b: C1 0010b: C3 0011b: C6 0100b: C7 0101b: C7S 0110b: C8 0111b: C9 1000b: C10	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.	
63:16	Reserved.	
Register Address: 17DH, 381	MSR_SMM_MCA_CAP	
Enhanced SMM Capabilities (SMM-RO) Reports SMM capability enhancement. Accessible only while in SMM.		Core
57:0	Reserved.	
58	SMM_Code_Access_Chk (SMM-RO) If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.	
59	Long_Flow_Indication (SMM-RO) If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.	
63:60	Reserved.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 188H, 392	IA32_PERFEVTSEL2	
See Table 2-2.		Core
Register Address: 189H, 393	IA32_PERFEVTSEL3	
See Table 2-2.		Core
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		
0	Fast-Strings Enable See Table 2-2.	Core
2:1	Reserved.	
3	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 1.	Package
6:4	Reserved.	
7	Performance Monitoring Available (R) See Table 2-2.	Core
10:8	Reserved.	
11	Branch Trace Storage Unavailable (R/O) See Table 2-2.	Core
12	Processor Event Based Sampling Unavailable (R/O) See Table 2-2.	Core
15:13	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.	Package
18	ENABLE MONITOR FSM (R/W) See Table 2-2.	Core
21:19	Reserved.	
22	Limit CPUID Maxval (R/W) See Table 2-2.	Core
23	xTPR Message Disable (R/W) See Table 2-2.	Package
33:24	Reserved.	
34	XD Bit Disable (R/W) See Table 2-3.	Core
37:35	Reserved.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
38	Turbo Mode Disable (R/W) When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.	Package
63:39	Reserved.	
Register Address: 1A4H, 420	MSR_MISC_FEATURE_CONTROL	
Miscellaneous Feature Control (R/W)		
0	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.	Core
1	Reserved.	
2	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.	Core
63:3	Reserved.	
Register Address: 1AAH, 426	MSR_MISC_PWR_MGMT	
Miscellaneous Power Management Control Various model specific features enumeration. See http://biosbits.org.		Package
0	EIST Hardware Coordination Disable (R/W) When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores. When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.	
21:1	Reserved.	
22	Thermal Interrupt Coordination Enable (R/W) If set, then thermal interrupt on one core is routed to all cores.	
63:23	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode by Core Groups (R/W) Specifies Maximum Ratio Limit for each Core Group. Max ratio for groups with more cores must decrease monotonically. For groups with less than 4 cores, the max ratio must be 32 or less. For groups with 4-5 cores, the max ratio must be 22 or less. For groups with more than 5 cores, the max ratio must be 16 or less.		Package
7:0	Maximum Ratio Limit for Active Cores in Group 0 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 0 threshold.	Package

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
15:8	Maximum Ratio Limit for Active Cores in Group 1 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 1 threshold, and greater than the Group 0 threshold.	Package
23:16	Maximum Ratio Limit for Active Cores in Group 2 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 2 threshold, and greater than the Group 1 threshold.	Package
31:24	Maximum Ratio Limit for Active Cores in Group 3 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 3 threshold, and greater than the Group 2 threshold.	Package
39:32	Maximum Ratio Limit for Active Cores in Group 4 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 4 threshold, and greater than the Group 3 threshold.	Package
47:40	Maximum Ratio Limit for Active Cores in Group 5 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 5 threshold, and greater than the Group 4 threshold.	Package
55:48	Maximum Ratio Limit for Active Cores in Group 6 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 6 threshold, and greater than the Group 5 threshold.	Package
63:56	Maximum Ratio Limit for Active Cores in Group 7 Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 7 threshold, and greater than the Group 6 threshold.	Package
Register Address: 1AEH, 430	MSR_TURBO_GROUP_CORECNT	
Group Size of Active Cores for Turbo Mode Operation (R/W) Writes of 0 threshold is ignored.		Package
7:0	Group 0 Core Count Threshold Maximum number of active cores to operate under the Group 0 Max Turbo Ratio limit.	Package
15:8	Group 1 Core Count Threshold Maximum number of active cores to operate under the Group 1 Max Turbo Ratio limit. Must be greater than the Group 0 Core Count.	Package
23:16	Group 2 Core Count Threshold Maximum number of active cores to operate under the Group 2 Max Turbo Ratio limit. Must be greater than the Group 1 Core Count.	Package
31:24	Group 3 Core Count Threshold Maximum number of active cores to operate under the Group 3 Max Turbo Ratio limit. Must be greater than the Group 2 Core Count.	Package
39:32	Group 4 Core Count Threshold Maximum number of active cores to operate under the Group 4 Max Turbo Ratio limit. Must be greater than the Group 3 Core Count.	Package
47:40	Group 5 Core Count Threshold Maximum number of active cores to operate under the Group 5 Max Turbo Ratio limit. Must be greater than the Group 4 Core Count.	Package

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
55:48	Group 6 Core Count Threshold Maximum number of active cores to operate under the Group 6 Max Turbo Ratio limit. Must be greater than the Group 5 Core Count.	Package
63:56	Group 7 Core Count Threshold Maximum number of active cores to operate under the Group 7 Max Turbo Ratio limit. Must be greater than the Group 6 Core Count, and not less than the total number of processor cores in the package. E.g., specify 255.	Package
Register Address: 1C8H, 456	MSR_LBR_SELECT	
Last Branch Record Filtering Select Register (R/W) See Section 19.9.2, “Filtering of Last Branch Records.”		Core
0	CPL_EQ_0	
1	CPL_NEQ_0	
2	JCC	
3	NEAR_REL_CALL	
4	NEAR_IND_CALL	
5	NEAR_RET	
6	NEAR_IND_JMP	
7	NEAR_REL_JMP	
8	FAR_BRANCH	
9	EN_CALL_STACK	
63:10	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-4) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 1FCH, 508	MSR_POWER_CTL	
Power Control Register See http://biosbits.org.		Core
0	Reserved.	
1	C1E Enable (R/W) When set to ‘1’, will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).	Package
63:2	Reserved.	
Register Address: 210H, 528	IA32_MTRR_PHYSBASE8	
See Table 2-2.		Core
Register Address: 211H, 529	IA32_MTRR_PHYSMASK8	
See Table 2-2.		Core
Register Address: 212H, 530	IA32_MTRR_PHYSBASE9	
See Table 2-2.		Core

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address:	IA32_MTRR_PHYSMASK9	
213H, 531	See Table 2-2.	Core
Register Address:	IA32_MC0_CTL2	
280H, 640	See Table 2-2.	Module
Register Address:	IA32_MC1_CTL2	
281H, 641	See Table 2-2.	Module
Register Address:	IA32_MC2_CTL2	
282H, 642	See Table 2-2.	Core
Register Address: 283H, 643	IA32_MC3_CTL2	
See Table 2-2.		Module
Register Address: 284H, 644	IA32_MC4_CTL2	
See Table 2-2.		Package
Register Address: 285H, 645	IA32_MC5_CTL2	
See Table 2-2.		Package
Register Address: 286H, 646	IA32_MC6_CTL2	
See Table 2-2.		Package
Register Address: 300H, 768	MSR_SGXOWNEREPOCH0	
Lower 64 Bit CR_SGXOWNEREPOCH (W) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.		Package
63:0	Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.	
Register Address: 301H, 769	MSR_SGXOWNEREPOCH1	
Upper 64 Bit CR_SGXOWNEREPOCH (W) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.		Package
63:0	Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.	
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2 and Section 21.2.4, “Architectural Performance Monitoring Version 4.”		Core
0	Ovf_PMC0	
1	Ovf_PMC1	
2	Ovf_PMC2	
3	Ovf_PMC3	
31:4	Reserved.	
32	Ovf_FixedCtr0	
33	Ovf_FixedCtr1	
34	Ovf_FixedCtr2	
54:35	Reserved.	
55	Trace_ToPA_PMI	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
57:56	Reserved.	
58	LBR_Frz	
59	CTR_Frz	
60	ASCI	
61	Ovf_Uncore	
62	Ovf_BufDSSAVE	
63	CondChgd	
Register Address: 390H, 912	IA32_PERF_GLOBAL_STATUS_RESET	
See Table 2-2 and Section 21.2.4, "Architectural Performance Monitoring Version 4."		Core
0	Set 1 to clear Ovf_PMC0.	
1	Set 1 to clear Ovf_PMC1.	
2	Set 1 to clear Ovf_PMC2.	
3	Set 1 to clear Ovf_PMC3.	
31:4	Reserved.	
32	Set 1 to clear Ovf_FixedCtr0.	
33	Set 1 to clear Ovf_FixedCtr1.	
34	Set 1 to clear Ovf_FixedCtr2.	
54:35	Reserved.	
55	Set 1 to clear Trace_ToPA_PMI.	
57:56	Reserved.	
58	Set 1 to clear LBR_Frz.	
59	Set 1 to clear CTR_Frz.	
60	Set 1 to clear ASCI.	
61	Set 1 to clear Ovf_Uncore.	
62	Set 1 to clear Ovf_BufDSSAVE.	
63	Set 1 to clear CondChgd.	
Register Address: 391H, 913	IA32_PERF_GLOBAL_STATUS_SET	
See Table 2-2 and Section 21.2.4, "Architectural Performance Monitoring Version 4."		Core
0	Set 1 to cause Ovf_PMC0 = 1.	
1	Set 1 to cause Ovf_PMC1 = 1.	
2	Set 1 to cause Ovf_PMC2 = 1.	
3	Set 1 to cause Ovf_PMC3 = 1.	
31:4	Reserved.	
32	Set 1 to cause Ovf_FixedCtr0 = 1.	
33	Set 1 to cause Ovf_FixedCtr1 = 1.	
34	Set 1 to cause Ovf_FixedCtr2 = 1.	
54:35	Reserved.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
55	Set 1 to cause Trace_ToPA_PMI = 1.	
57:56	Reserved.	
58	Set 1 to cause LBR_Frz = 1.	
59	Set 1 to cause CTR_Frz = 1.	
60	Set 1 to cause ASCI = 1.	
61	Set 1 to cause Ovf_Uncore.	
62	Set 1 to cause Ovf_BufDSSAVE.	
63	Reserved.	
Register Address: 392H, 914	IA32_PERF_GLOBAL_INUSE	
See Table 2-2.		Core
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Table 2-2 and Section 21.6.2.4, "Processor Event Based Sampling (PEBS)."		Core
0	Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0. (R/W)	
Register Address: 3F8H, 1016	MSR_PKG_C3_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C3 Residency Counter (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.	
Register Address: 3F9H, 1017	MSR_PKG_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C6 Residency Counter (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.	
Register Address: 3FCH, 1020	MSR_CORE_C3_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C3 Residency Counter (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.	
Register Address: 406H, 1030	IA32_MC1_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Module
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 4C3H, 1219	IA32_A_PMC2	
See Table 2-2.		Core
Register Address: 4C4H, 1220	IA32_A_PMC3	
See Table 2-2.		Core
Register Address: 4E0H, 1248	MSR_SMM_FEATURE_CONTROL	
Enhanced SMM Feature Control (SMM-RW) Reports SMM capability Enhancement. Accessible only while in SMM.		Package
0	Lock (SMM-RWO) When set to '1' locks this register from further changes.	
1	Reserved.	
2	SMM_Code_Chk_En (SMM-RW) This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.	
63:3	Reserved.	
Register Address: 4E2H, 1250	MSR_SMM_DELAYED	
SMM Delayed (SMM-RO) Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.		Package
N-1:0	LOG_PROC_STATE (SMM-RO) Each bit represents a processor core of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.	
63:N	Reserved.	
Register Address: 4E3H, 1251	MSR_SMM_BLOCKED	
SMM Blocked (SMM-RO) Reports the blocked state of all logical processors in the package. Available only while in SMM.		Package

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
N-1:0	LOG_PROC_STATE (SMM-RO) Each bit represents a processor core of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.	
63:N	Reserved.	
Register Address: 500H, 1280	IA32_SGX_SVN_STATUS	
Status and SVN Threshold of SGX Support for ACM (R/O)		Core
0	Lock See Section 40.11.3, "Interactions with Authenticated Code Modules (ACMs)."	
15:1	Reserved.	
23:16	SGX_SVN_SINIT See Section 40.11.3, "Interactions with Authenticated Code Modules (ACMs)."	
63:24	Reserved.	
Register Address: 560H, 1376	IA32_RTIT_OUTPUT_BASE	
Trace Output Base Register (R/W) See Table 2-2.		Core
Register Address: 561H, 1377	IA32_RTIT_OUTPUT_MASK_PTRS	
Trace Output Mask Pointers Register (R/W) See Table 2-2.		Core
Register Address: 570H, 1392	IA32_RTIT_CTL	
Trace Control Register (R/W)		Core
0	TraceEn	
1	CYCEn	
2	OS	
3	User	
6:4	Reserved, must be zero.	
7	CR3Filter	
8	ToPA Writing 0 will #GP if also setting TraceEn.	
9	MTCEn	
10	TSCEn	
11	DisRETC	
12	Reserved, must be zero.	
13	BranchEn	
17:14	MTCFreq	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
18	Reserved, must be zero.	
22:19	CycThresh	
23	Reserved, must be zero.	
27:24	PSBFreq	
31:28	Reserved, must be zero.	
35:32	ADDR0_CFG	
39:36	ADDR1_CFG	
63:40	Reserved, must be zero.	
Register Address: 571H, 1393	IA32_RTIT_STATUS	
Tracing Status Register (R/W)		Core
0	FilterEn Writes ignored.	
1	ContextEn Writes ignored.	
2	TriggerEn Writes ignored.	
3	Reserved	
4	Error (R/W)	
5	Stopped	
31:6	Reserved, must be zero.	
48:32	PacketByteCnt	
63:49	Reserved, must be zero.	
Register Address: 572H, 1394	IA32_RTIT_CR3_MATCH	
Trace Filter CR3 Match Register (R/W)		Core
4:0	Reserved	
63:5	CR3[63:5] value to match.	
Register Address: 580H, 1408	IA32_RTIT_ADDR0_A	
Region 0 Start Address (R/W)		Core
63:0	See Table 2-2.	
Register Address: 581H, 1409	IA32_RTIT_ADDR0_B	
Region 0 End Address (R/W)		Core
63:0	See Table 2-2.	
Register Address: 582H, 1410	IA32_RTIT_ADDR1_A	
Region 1 Start Address (R/W)		Core
63:0	See Table 2-2.	
Register Address: 583H, 1411	IA32_RTIT_ADDR1_B	
Region 1 End Address (R/W)		Core
63:0	See Table 2-2.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers used in RAPL Interfaces (R/O) See Section 16.10.1, "RAPL Interfaces."		Package
3:0	Power Units Power related information (in Watts) is in unit of 1W/2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 1000b, indicating power unit is in 3.9 milliWatts increment.	
7:4	Reserved.	
12:8	Energy Status Units Energy related information (in Joules) is in unit of 1Joule/ (2^ESU); where ESU is an unsigned integer represented by bits 12:8. Default value is 01110b, indicating energy unit is in 61 microJoules.	
15:13	Reserved.	
19:16	Time Unit Time related information (in seconds) is in unit of 1S/2^TU; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating power unit is in 0.977 millisecond.	
63:20	Reserved.	
Register Address: 60AH, 1546	MSR_PKGC3_IRTL	
Package C3 Interrupt Response Limit (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
9:0	Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C3 state.	
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	
63:16	Reserved.	
Register Address: 60BH, 1547	MSR_PKGC_IRTL1	
Package C6/C7S Interrupt Response Limit 1 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7S state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Package
9:0	Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	
63:16	Reserved.	
Register Address: 60CH, 1548	MSR_PKGC_IRTL2	
Package C7 Interrupt Response Limit 2 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
9:0	Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C7 state.	
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	
63:16	Reserved.	
Register Address: 60DH, 1549	MSR_PKG_C2_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Package
63:0	Package C2 Residency Counter (R/O) Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.	
Register Address: 610H, 1552	MSR_PKG_POWER_LIMIT	
PKG RAPL Power Limit Control (R/W) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 611H, 1553	MSR_PKG_ENERGY_STATUS	
PKG Energy Status (R/O) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 613H, 1555	MSR_PKG_PERF_STATUS	
PKG Perf Status (R/O) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 614H, 1556	MSR_PKG_POWER_INFO	
PKG RAPL Parameters (R/W)		Package

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
14:0	Thermal Spec Power (R/W) See Section 16.10.3, "Package RAPL Domain."	
15	Reserved.	
30:16	Minimum Power (R/W) See Section 16.10.3, "Package RAPL Domain."	
31	Reserved.	
46:32	Maximum Power (R/W) See Section 16.10.3, "Package RAPL Domain."	
47	Reserved.	
54:48	Maximum Time Window (R/W) Specified by 2^Y * (1.0 + Z/4.0) * Time_Unit, where "Y" is the unsigned integer value represented by bits 52:48, "Z" is an unsigned integer represented by bits 54:53. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.	
63:55	Reserved.	
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
DRAM RAPL Power Limit Control (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	
DRAM RAPL Parameters (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 632H, 1586	MSR_PKG_C10_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Package
63:0	Package C10 Residency Counter (R/O) Value since last reset that the entire SOC is in an S0i3 state. Count at the same frequency as the TSC.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 641H, 1601	MSR_PP1_ENERGY_STATUS	
PP1 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 64CH, 1612	MSR_TURBO_ACTIVATION_RATIO	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
ConfigTDP Control (R/W)		Package
7:0	MAX_NON_TURBO_RATIO (RW/L) System BIOS can program this field.	
30:8	Reserved.	
31	TURBO_ACTIVATION_RATIO_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.	
63:32	Reserved.	
Register Address: 64FH, 1615	MSR_CORE_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)		Package
0	PROCHOT Status (R0) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.	
2	Package-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.	
3	Package-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.	
8:4	Reserved.	
9	Core Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to domain-level power limiting.	
10	VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.	
11	Max Turbo Limit Status (R0) When set, frequency is reduced below the operating system request due to multi-core turbo limits.	
12	Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).	
13	Turbo Transition Attenuation Status (R0) When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.	
14	Maximum Efficiency Frequency Status (R0) When set, frequency is reduced below the maximum efficiency frequency.	
15	Reserved.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
18	Package-Level PL1 Power Limiting Log When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19	Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
24:20	Reserved.	
25	Core Power Limiting Log When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
26	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
27	Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
29	Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
30	Maximum Efficiency Frequency Log When set, indicates that the Maximum Efficiency Frequency Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:31	Reserved.	
Register Address: 680H, 1664	MSR_LASTBRANCH_0_FROM_IP	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 0 From IP (R/W) One of 32 pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.6 and record format in Section 19.4.8.1.		Core
0:47	From Linear Address (R/W)	
62:48	Signed extension of bits 47:0.	
63	Mispred	
Register Address: 681H, 1665	MSR_LASTBRANCH_1_FROM_IP	
Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 682H, 1666	MSR_LASTBRANCH_2_FROM_IP	
Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 683H, 1667	MSR_LASTBRANCH_3_FROM_IP	
Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 684H, 1668	MSR_LASTBRANCH_4_FROM_IP	
Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 685H, 1669	MSR_LASTBRANCH_5_FROM_IP	
Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 686H, 1670	MSR_LASTBRANCH_6_FROM_IP	
Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 687H, 1671	MSR_LASTBRANCH_7_FROM_IP	
Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 688H, 1672	MSR_LASTBRANCH_8_FROM_IP	
Last Branch Record 8 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 689H, 1673	MSR_LASTBRANCH_9_FROM_IP	
Last Branch Record 9 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 68AH, 1674	MSR_LASTBRANCH_10_FROM_IP	
Last Branch Record 10 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 68BH, 1675	MSR_LASTBRANCH_11_FROM_IP	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 11 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 68CH, 1676	MSR_LASTBRANCH_12_FROM_IP	
Last Branch Record 12 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 68DH, 1677	MSR_LASTBRANCH_13_FROM_IP	
Last Branch Record 13 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 68EH, 1678	MSR_LASTBRANCH_14_FROM_IP	
Last Branch Record 14 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 68FH, 1679	MSR_LASTBRANCH_15_FROM_IP	
Last Branch Record 15 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 690H, 1680	MSR_LASTBRANCH_16_FROM_IP	
Last Branch Record 16 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 691H, 1681	MSR_LASTBRANCH_17_FROM_IP	
Last Branch Record 17 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 692H, 1682	MSR_LASTBRANCH_18_FROM_IP	
Last Branch Record 18 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 693H, 1683	MSR_LASTBRANCH_19_FROM_IP	
Last Branch Record 19From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 694H, 1684	MSR_LASTBRANCH_20_FROM_IP	
Last Branch Record 20 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 695H, 1685	MSR_LASTBRANCH_21_FROM_IP	
Last Branch Record 21 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 696H, 1686	MSR_LASTBRANCH_22_FROM_IP	
Last Branch Record 22 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 697H, 1687	MSR_LASTBRANCH_23_FROM_IP	
Last Branch Record 23 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 698H, 1688	MSR_LASTBRANCH_24_FROM_IP	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 24 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 699H, 1689	MSR_LASTBRANCH_25_FROM_IP	
Last Branch Record 25 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 69AH, 1690	MSR_LASTBRANCH_26_FROM_IP	
Last Branch Record 26 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 69BH, 1691	MSR_LASTBRANCH_27_FROM_IP	
Last Branch Record 27 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 69CH, 1692	MSR_LASTBRANCH_28_FROM_IP	
Last Branch Record 28 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 69DH, 1693	MSR_LASTBRANCH_29_FROM_IP	
Last Branch Record 29 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 69EH, 1694	MSR_LASTBRANCH_30_FROM_IP	
Last Branch Record 30 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 69FH, 1695	MSR_LASTBRANCH_31_FROM_IP	
Last Branch Record 31 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Core
Register Address: 6C0H, 1728	MSR_LASTBRANCH_0_TO_IP	
Last Branch Record 0 To IP (R/W) One of 32 pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the Destination instruction and elapsed cycles from last LBR update. See Section 19.6.		Core
0:47	Target Linear Address (R/W)	
63:48	Elapsed cycles from last update to the LBR.	
Register Address: 6C1H, 1729	MSR_LASTBRANCH_1_TO_IP	
Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C2H, 1730	MSR_LASTBRANCH_2_TO_IP	
Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C3H, 1731	MSR_LASTBRANCH_3_TO_IP	
Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C4H, 1732	MSR_LASTBRANCH_4_TO_IP	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C5H, 1733	MSR_LASTBRANCH_5_TO_IP	
Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C6H, 1734	MSR_LASTBRANCH_6_TO_IP	
Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C7H, 1735	MSR_LASTBRANCH_7_TO_IP	
Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C8H, 1736	MSR_LASTBRANCH_8_TO_IP	
Last Branch Record 8 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6C9H, 1737	MSR_LASTBRANCH_9_TO_IP	
Last Branch Record 9 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6CAH, 1738	MSR_LASTBRANCH_10_TO_IP	
Last Branch Record 10 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6CBH, 1739	MSR_LASTBRANCH_11_TO_IP	
Last Branch Record 11 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6CCH, 1740	MSR_LASTBRANCH_12_TO_IP	
Last Branch Record 12 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6CDH, 1741	MSR_LASTBRANCH_13_TO_IP	
Last Branch Record 13 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6CEH, 1742	MSR_LASTBRANCH_14_TO_IP	
Last Branch Record 14 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6CFH, 1743	MSR_LASTBRANCH_15_TO_IP	
Last Branch Record 15 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D0H, 1744	MSR_LASTBRANCH_16_TO_IP	
Last Branch Record 16 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D1H, 1745	MSR_LASTBRANCH_17_TO_IP	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 17 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D2H, 1746	MSR_LASTBRANCH_18_TO_IP	
Last Branch Record 18 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D3H, 1747	MSR_LASTBRANCH_19_TO_IP	
Last Branch Record 19To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D4H, 1748	MSR_LASTBRANCH_20_TO_IP	
Last Branch Record 20 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D5H, 1749	MSR_LASTBRANCH_21_TO_IP	
Last Branch Record 21 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D6H, 1750	MSR_LASTBRANCH_22_TO_IP	
Last Branch Record 22 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D7H, 1751	MSR_LASTBRANCH_23_TO_IP	
Last Branch Record 23 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D8H, 1752	MSR_LASTBRANCH_24_TO_IP	
Last Branch Record 24 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6D9H, 1753	MSR_LASTBRANCH_25_TO_IP	
Last Branch Record 25 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6DAH, 1754	MSR_LASTBRANCH_26_TO_IP	
Last Branch Record 26 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6DBH, 1755	MSR_LASTBRANCH_27_TO_IP	
Last Branch Record 27 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6DCH, 1756	MSR_LASTBRANCH_28_TO_IP	
Last Branch Record 28 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6DDH, 1757	MSR_LASTBRANCH_29_TO_IP	
Last Branch Record 29 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6DEH, 1758	MSR_LASTBRANCH_30_TO_IP	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 30 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 6DFH, 1759	MSR_LASTBRANCH_31_TO_IP	
Last Branch Record 31 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Core
Register Address: 802H, 2050	IA32_X2APIC_APICID	
x2APIC ID register (R/O)		Core
Register Address: 803H, 2051	IA32_X2APIC_VERSION	
x2APIC Version register (R/O)		Core
Register Address: 808H, 2056	IA32_X2APIC_TPR	
x2APIC Task Priority register (R/W)		Core
Register Address: 80AH, 2058	IA32_X2APIC_PPR	
x2APIC Processor Priority register (R/O)		Core
Register Address: 80BH, 2059	IA32_X2APIC_EOI	
x2APIC EOI register (W/O)		Core
Register Address: 80DH, 2061	IA32_X2APIC_LDR	
x2APIC Logical Destination register (R/O)		Core
Register Address: 80FH, 2063	IA32_X2APIC_SIVR	
x2APIC Spurious Interrupt Vector register (R/W)		Core
Register Address: 810H, 2064	IA32_X2APIC_ISR0	
x2APIC In-Service register bits [31:0] (R/O)		Core
Register Address: 811H, 2065	IA32_X2APIC_ISR1	
x2APIC In-Service register bits [63:32] (R/O)		Core
Register Address: 812H, 2066	IA32_X2APIC_ISR2	
x2APIC In-Service register bits [95:64] (R/O)		Core
Register Address: 813H, 2067	IA32_X2APIC_ISR3	
x2APIC In-Service register bits [127:96] (R/O)		Core
Register Address: 814H, 2068	IA32_X2APIC_ISR4	
x2APIC In-Service register bits [159:128] (R/O)		Core
Register Address: 815H, 2069	IA32_X2APIC_ISR5	
x2APIC In-Service register bits [191:160] (R/O)		Core
Register Address: 816H, 2070	IA32_X2APIC_ISR6	
x2APIC In-Service register bits [223:192] (R/O)		Core
Register Address: 817H, 2071	IA32_X2APIC_ISR7	
x2APIC In-Service register bits [255:224] (R/O)		Core
Register Address: 818H, 2072	IA32_X2APIC_TMR0	
x2APIC Trigger Mode register bits [31:0] (R/O)		Core
Register Address: 819H, 2073	IA32_X2APIC_TMR1	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
x2APIC Trigger Mode register bits [63:32] (R/O)		Core
Register Address: 81AH, 2074	IA32_X2APIC_TMR2	
x2APIC Trigger Mode register bits [95:64] (R/O)		Core
Register Address: 81BH, 2075	IA32_X2APIC_TMR3	
x2APIC Trigger Mode register bits [127:96] (R/O)		Core
Register Address: 81CH, 2076	IA32_X2APIC_TMR4	
x2APIC Trigger Mode register bits [159:128] (R/O)		Core
Register Address: 81DH, 2077	IA32_X2APIC_TMR5	
x2APIC Trigger Mode register bits [191:160] (R/O)		Core
Register Address: 81EH, 2078	IA32_X2APIC_TMR6	
x2APIC Trigger Mode register bits [223:192] (R/O)		Core
Register Address: 81FH, 2079	IA32_X2APIC_TMR7	
x2APIC Trigger Mode register bits [255:224] (R/O)		Core
Register Address: 820H, 2080	IA32_X2APIC_IRR0	
x2APIC Interrupt Request register bits [31:0] (R/O)		Core
Register Address: 821H, 2081	IA32_X2APIC_IRR1	
x2APIC Interrupt Request register bits [63:32] (R/O)		Core
Register Address: 822H, 2082	IA32_X2APIC_IRR2	
x2APIC Interrupt Request register bits [95:64] (R/O)		Core
Register Address: 823H, 2083	IA32_X2APIC_IRR3	
x2APIC Interrupt Request register bits [127:96] (R/O)		Core
Register Address: 824H, 2084	IA32_X2APIC_IRR4	
x2APIC Interrupt Request register bits [159:128] (R/O)		Core
Register Address: 825H, 2085	IA32_X2APIC_IRR5	
x2APIC Interrupt Request register bits [191:160] (R/O)		Core
Register Address: 826H, 2086	IA32_X2APIC_IRR6	
x2APIC Interrupt Request register bits [223:192] (R/O)		Core
Register Address: 827H, 2087	IA32_X2APIC_IRR7	
x2APIC Interrupt Request register bits [255:224] (R/O)		Core
Register Address: 828H, 2088	IA32_X2APIC_ESR	
x2APIC Error Status register (R/W)		Core
Register Address: 82FH, 2095	IA32_X2APIC_LVT_CMCI	
x2APIC LVT Corrected Machine Check Interrupt register (R/W)		Core
Register Address: 830H, 2096	IA32_X2APIC_ICR	
x2APIC Interrupt Command register (R/W)		Core
Register Address: 832H, 2098	IA32_X2APIC_LVT_TIMER	
x2APIC LVT Timer Interrupt register (R/W)		Core

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 833H, 2099	IA32_X2APIC_LVT_THERMAL	
x2APIC LVT Thermal Sensor Interrupt register (R/W)		Core
Register Address: 834H, 2100	IA32_X2APIC_LVT_PMI	
x2APIC LVT Performance Monitor register (R/W)		Core
Register Address: 835H, 2101	IA32_X2APIC_LVT_LINT0	
x2APIC LVT LINT0 register (R/W)		Core
Register Address: 836H, 2102	IA32_X2APIC_LVT_LINT1	
x2APIC LVT LINT1 register (R/W)		Core
Register Address: 837H, 2103	IA32_X2APIC_LVT_ERROR	
x2APIC LVT Error register (R/W)		Core
Register Address: 838H, 2104	IA32_X2APIC_INIT_COUNT	
x2APIC Initial Count register (R/W)		Core
Register Address: 839H, 2105	IA32_X2APIC_CUR_COUNT	
x2APIC Current Count register (R/O)		Core
Register Address: 83EH, 2110	IA32_X2APIC_DIV_CONF	
x2APIC Divide Configuration register (R/W)		Core
Register Address: 83FH, 2111	IA32_X2APIC_SELF_IPI	
x2APIC Self IPI register (W/O)		Core
Register Address: C8FH, 3215	IA32_PQR_ASSOC	
Resource Association Register (R/W)		Core
31:0	Reserved.	
33:32	CLOS (R/W)	
63: 34	Reserved.	
Register Address: D10H, 3344	IA32_L2_QOS_MASK_0	
L2 Class Of Service Mask - CLOS 0 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0.		Module
0:7	CBM: Bit vector of available L2 ways for CLOS 0 enforcement.	
63:8	Reserved.	
Register Address: D11H, 3345	IA32_L2_QOS_MASK_1	
L2 Class Of Service Mask - CLOS 1 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1.		Module
0:7	CBM: Bit vector of available L2 ways for CLOS 0 enforcement.	
63:8	Reserved.	
Register Address: D12H, 3346	IA32_L2_QOS_MASK_2	
L2 Class Of Service Mask - CLOS 2 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2.		Module
0:7	CBM: Bit vector of available L2 ways for CLOS 0 enforcement.	
63:8	Reserved.	

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: D13H, 3347	IA32_L2_QOS_MASK_3	
L2 Class Of Service Mask - CLOS 3 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3.		Package
0:19	CBM: Bit vector of available L2 ways for CLOS 3 enforcement.	
63:20	Reserved.	
Register Address: D90H, 3472	IA32_BNDCFGS	
See Table 2-2.		Core
Register Address: DA0H, 3488	IA32_XSS	
See Table 2-2.		Core
See Table 2-6, and Table 2-12 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_5CH.		

2.6 MSRS IN INTEL ATOM® PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Intel Atom processors based on the Goldmont Plus microarchitecture support MSRs listed in Table 2-6, Table 2-12, and Table 2-13. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_7AH; see Table 2-1. For an MSR listed in Table 2-13 that also appears in the model-specific tables of prior generations, Table 2-13 supersedes prior generation tables.

In the Goldmont Plus microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Goldmont Plus microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64Processor (R/W) See Table 2-2.		Core
0	Lock (R/WL)	
1	Enable VMX inside SMX operation (R/WL)	
2	Enable VMX outside SMX operation (R/WL)	
14:8	SENTER local functions enables (R/WL)	
15	SENTER global functions enable (R/WL)	
17	SGX Launch Control Enable (R/WL) This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Valid if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.	

Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
18	SGX global functions enable (R/WL)	
63:19	Reserved.	
Register Address: 8CH, 140	IA32_SGXLEPUBKEYHASH0	
See Table 2-2.		Core
Register Address: 8DH, 141	IA32_SGXLEPUBKEYHASH1	
See Table 2-2.		Core
Register Address: 8EH, 142	IA32_SGXLEPUBKEYHASH2	
See Table 2-2.		Core
Register Address: 8FH, 143	IA32_SGXLEPUBKEYHASH3	
See Table 2-2.		Core
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
(R/W) See Table 2-2. See Section 21.6.2.4, “Processor Event Based Sampling (PEBS).”		Core
0	Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0.	
1	Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC1.	
2	Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC2.	
3	Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC3.	
31:4	Reserved.	
32	Enable PEBS trigger and recording for IA32_FIXED_CTR0.	
33	Enable PEBS trigger and recording for IA32_FIXED_CTR1.	
34	Enable PEBS trigger and recording for IA32_FIXED_CTR2.	
63:35	Reserved.	
Register Address: 570H, 1392	IA32_RTIT_CTL	
Trace Control Register (R/W)		Core
0	TraceEn	
1	CYCEn	
2	OS	
3	User	
4	PwrEvtEn	
5	FUPonPTW	
6	FabricEn	
7	CR3Filter	
8	ToPA Writing 0 will #GP if also setting TraceEn.	
9	MTCEn	
10	TSCEn	

Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
11	DisRETC	
12	PTWEn	
13	BranchEn	
17:14	MTCFreq	
18	Reserved, must be zero.	
22:19	CycThresh	
23	Reserved, must be zero.	
27:24	PSBFreq	
31:28	Reserved, must be zero.	
35:32	ADDR0_CFG	
39:36	ADDR1_CFG	
63:40	Reserved, must be zero.	
Register Address: 680H, 1664	MSR_LASTBRANCH_0_FROM_IP	
Last Branch Record 0 From IP (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. The From_IP part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."		Core
Register Address: 681H—69FH, 1665—1695	MSR_LASTBRANCH_*i*_FROM_IP	
Last Branch Record *i* From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP; *i* = 1-31.		Core
Register Address: 6C0H, 1728	MSR_LASTBRANCH_0_TO_IP	
Last Branch Record 0 To IP (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. The To_IP part of the stack contains pointers to the Destination instruction. See also: ▪ Section 19.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."		Core
Register Address: 6C1H—6DFH, 1729—1759	MSR_LASTBRANCH_*i*_TO_IP	
Last Branch Record *i* To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP; *i* = 1-31.		Core
Register Address: DC0H, 3520	MSR_LASTBRANCH_INFO_0	
Last Branch Record 0 Additional Information (R/W) One of the three MSRs that make up the first entry of the 32-entry LBR stack. This part of the stack contains flag and elapsed cycle information. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.9.1, "LBR Stack."		Core
Register Address: DC1H, 3521	MSR_LASTBRANCH_INFO_1	
Last Branch Record 1 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core

Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: DC2H, 3522	MSR_LASTBRANCH_INFO_2	
Last Branch Record 2 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DC3H, 3523	MSR_LASTBRANCH_INFO_3	
Last Branch Record 3 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DC4H, 3524	MSR_LASTBRANCH_INFO_4	
Last Branch Record 4 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DC5H, 3525	MSR_LASTBRANCH_INFO_5	
Last Branch Record 5 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DC6H, 3526	MSR_LASTBRANCH_INFO_6	
Last Branch Record 6 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DC7H, 3527	MSR_LASTBRANCH_INFO_7	
Last Branch Record 7 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DC8H, 3528	MSR_LASTBRANCH_INFO_8	
Last Branch Record 8 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DC9H, 3529	MSR_LASTBRANCH_INFO_9	
Last Branch Record 9 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DCAH, 3530	MSR_LASTBRANCH_INFO_10	
Last Branch Record 10 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DCBH, 3531	MSR_LASTBRANCH_INFO_11	
Last Branch Record 11 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DCCH, 3532	MSR_LASTBRANCH_INFO_12	
Last Branch Record 12 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DCDH, 3533	MSR_LASTBRANCH_INFO_13	
Last Branch Record 13 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DCEH, 3534	MSR_LASTBRANCH_INFO_14	
Last Branch Record 14 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core

Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: DCFH, 3535	MSR_LASTBRANCH_INFO_15	
Last Branch Record 15 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD0H, 3536	MSR_LASTBRANCH_INFO_16	
Last Branch Record 16 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD1H, 3537	MSR_LASTBRANCH_INFO_17	
Last Branch Record 17 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD2H, 3538	MSR_LASTBRANCH_INFO_18	
Last Branch Record 18 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD3H, 3539	MSR_LASTBRANCH_INFO_19	
Last Branch Record 19 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD4H, 3520	MSR_LASTBRANCH_INFO_20	
Last Branch Record 20 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD5H, 3521	MSR_LASTBRANCH_INFO_21	
Last Branch Record 21 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD6H, 3522	MSR_LASTBRANCH_INFO_22	
Last Branch Record 22 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD7H, 3523	MSR_LASTBRANCH_INFO_23	
Last Branch Record 23 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD8H, 3524	MSR_LASTBRANCH_INFO_24	
Last Branch Record 24 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DD9H, 3525	MSR_LASTBRANCH_INFO_25	
Last Branch Record 25 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DDAH, 3526	MSR_LASTBRANCH_INFO_26	
Last Branch Record 26 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DDBH, 3527	MSR_LASTBRANCH_INFO_27	
Last Branch Record 27 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core

Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: DDCH, 3528	MSR_LASTBRANCH_INFO_28	
Last Branch Record 28 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DDDH, 3529	MSR_LASTBRANCH_INFO_29	
Last Branch Record 29 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DDEH, 3530	MSR_LASTBRANCH_INFO_30	
Last Branch Record 30 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
Register Address: DDFH, 3531	MSR_LASTBRANCH_INFO_31	
Last Branch Record 31 Additional Information (R/W) See description of MSR_LASTBRANCH_INFO_0.		Core
See Table 2-6, Table 2-12, and Table 2-13 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_7AH.		

2.7 MSRS IN INTEL ATOM® PROCESSORS BASED ON TREMONT MICROARCHITECTURE

Processors based on the Tremont microarchitecture support MSRs listed in Table 2-6, Table 2-12, Table 2-13, and Table 2-14. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_86H, 06_96H, or 06_9CH; see Table 2-1. For an MSR listed in Table 2-14 that also appears in the model-specific tables of prior generations, Table 2-14 supersedes prior generation tables.

In the Tremont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Tremont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-14. MSRs in Intel Atom® Processors Based on Tremont Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 33H, 51	MSR_MEMORY_CTRL	
Memory Control Register		Core
28:0	Reserved.	
29	SPLIT_LOCK_DISABLE If set to 1, a split lock will cause an #AC(0) exception. See Section 10.1.2.3, "Features to Disable Bus Locks."	
30	Reserved.	
31	Reserved.	
Register Address: CFH, 207	IA32_CORE_CAPABILITIES	

Table 2-14. MSRs in Intel Atom® Processors Based on Tremont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32 Core Capabilities Register If CPUID.(EAX=07H, ECX=0):EDX[30] = 1.		Core
4:0	Reserved.	
5	SPLIT_LOCK_DISABLE_SUPPORTED When read as 1, software can set bit 29 of MSR_MEMORY_CTRL (MSR address 33H).	
63:6	Reserved.	
Register Address: 2A0H, 672	MSR_PRMRR_BASE_0	
Processor Reserved Memory Range Register - Physical Base Control Register (R/W)		Core
2:0	MEMTYPE: PRMRR BASE Memory Type.	
3	CONFIGURED: PRMRR BASE Configured.	
11:4	Reserved.	
51:12	BASE: PRMRR Base Address.	
63:52	Reserved.	
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
(R/W) See Table 2-2. See Section 21.6.2.4, “Processor Event Based Sampling (PEBS).”		Core
n:0	Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMCx. The maximum value n can be determined from CPUID.0AH:EAX[15:8].	
31:*n*+1	Reserved.	
32+*m*:32	Enable PEBS trigger and recording for IA32_FIXED_CTRx. The maximum value m can be determined from CPUID.0AH:EDX[4:0].	
59:33+*m*	Reserved.	
60	Pend a PerfMon Interrupt (PMI) after each PEBS event.	
62:61	Specifies PEBS output destination. Encodings: 00B: DS Save Area. 01B: Intel PT trace output. Supported if IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16] and CPUID.07H.0.EBX[25] are set. 10B: Reserved. 11B: Reserved.	
63	Reserved.	
Register Address: 1309H—130BH, 4873—4875	MSR_RELOAD_FIXED_CTRx	
Reload value for IA32_FIXED_CTRx (R/W)		
47:0	Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and FIXED_CTRx is overflowed.	
63:48	Reserved.	
Register Address: 14C1H—14C4H, 5313—5316	MSR_RELOAD_PMCx	

Table 2-14. MSRs in Intel Atom® Processors Based on Tremont Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Reload value for IA32_PMCx (R/W)		Core
47:0	Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and PMCx is overflowed.	
63:48	Reserved.	
See Table 2-6, Table 2-12, Table 2-13, and Table 2-14 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_86H.		

2.8 MSRS IN PROCESSORS BASED ON NEHALEM MICROARCHITECTURE

Table 2-15 lists model-specific registers (MSRs) that are common for Nehalem microarchitecture. These include the Intel Core i7 and i5 processor family. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_1AH, 06_1EH, 06_1FH, or 06_2EH; see Table 2-1. Additional MSRs specific to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_1AH, 06_1EH, or 06_1FH are listed in Table 2-16. Some MSRs listed in these tables are used by BIOS. More information about these MSR can be found at http://biosbits.org.

The column "Scope" represents the package/core/thread scope of individual bit field of an MSR. "Thread" means this bit field must be programmed on each logical processor independently. "Core" means the bit field must be programmed on each processor core independently, logical processors in the same core will be affected by change of this bit on the other logical processor in the same core. "Package" means the bit field must be programmed once for each physical package. Change of a bit filed with a package scope will affect all logical processors in that physical package.

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 0H, 0	IA32_P5_MC_ADDR	
See Section 2.23, "MSRs in Pentium Processors."		Thread
Register Address: 1H, 1	IA32_P5_MC_TYPE	
See Section 2.23, "MSRs in Pentium Processors."		Thread
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.		Thread
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER	
See Section 19.17, "Time-Stamp Counter," and Table 2-2.		Thread
Register Address: 17H, 23	IA32_PLATFORM_ID	
Platform ID (R) See Table 2-2.		Package
Register Address: 17H, 23	MSR_PLATFORM_ID	
Model Specific Platform ID (R)		Package
49:0	Reserved.	
52:50	See Table 2-2.	
63:53	Reserved.	
Register Address: 1BH, 27	IA32_APIC_BASE	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 12.4.4, "Local APIC Status and Location," and Table 2-2.		Thread
Register Address: 34H, 52	MSR_SMI_COUNT	
SMI Counter (R/O)		Thread
31:0	SMI Count (R/O) Running count of SMI events since last RESET.	
63:32	Reserved.	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64Processor (R/W) See Table 2-2.		Thread
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG	
BIOS Update Trigger Register (W) See Table 2-2.		Core
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Thread
Register Address: C1H, 193	IA32_PMC0	
Performance Counter Register See Table 2-2.		Thread
Register Address: C2H, 194	IA32_PMC1	
Performance Counter Register See Table 2-2.		Thread
Register Address: C3H, 195	IA32_PMC2	
Performance Counter Register See Table 2-2.		Thread
Register Address: C4H, 196	IA32_PMC3	
Performance Counter Register See Table 2-2.		Thread
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.	Package
27:16	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.	Package

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
29	Programmable TDC-TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDC and TDP Limits for Turbo mode are programmable. When set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.	Package
39:30	Reserved.	
47:40	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 133.33MHz.	Package
63:48	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.		Core
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 010b: C3 011b: C6 100b: C7 101b and 110b: Reserved 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.	
23:16	Reserved.	
24	Interrupt filtering enable (R/W) When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.	
25	C3 state auto demotion enable (R/W) When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
26	C1 state auto demotion enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.	
27	Enable C3 Undemotion (R/W)	
28	Enable C1 Undemotion (R/W)	
29	Package C State Demotion Enable (R/W)	
30	Package C State Undemotion Enable (R/W)	
63:31	Reserved.	
Register Address: E4H, 228	MSR_PMG_IO_CAPTURE_BASE	
Power Management IO Redirection in C-state (R/W) See http://biosbits.org.		Core
15:0	LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.	
18:16	C-state Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include. 001b - C6 is the max C-State to include. 010b - C7 is the max C-State to include.	
63:19	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: FEH, 254	IA32_MTRRCAP	
See Table 2-2.		Thread
Register Address: 174H, 372	IA32_SYSENTER_CS	
See Table 2-2.		Thread
Register Address: 175H, 373	IA32_SYSENTER_ESP	
See Table 2-2.		Thread
Register Address: 176H, 374	IA32_SYSENTER_EIP	
See Table 2-2.		Thread
Register Address: 179H, 377	IA32_MCG_CAP	
See Table 2-2.		Thread

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 17AH, 378	IA32_MCG_STATUS	
Global Machine Check Status		Thread
0	RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.	
1	EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.	
2	MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.	
63:3	Reserved.	
Register Address: 186H, 390	IA32_PERFEVTSEL0	
See Table 2-2.		Thread
7:0	Event Select	
15:8	UMask	
16	USR	
17	OS	
18	Edge	
19	PC	
20	INT	
21	AnyThread	
22	EN	
23	INV	
31:24	CMASK	
63:32	Reserved.	
Register Address: 187H, 391	IA32_PERFEVTSEL1	
See Table 2-2.		Thread
Register Address: 188H, 392	IA32_PERFEVTSEL2	
See Table 2-2.		Thread
Register Address: 189H, 393	IA32_PERFEVTSEL3	
See Table 2-2.		Thread
Register Address: 198H, 408	IA32_PERF_STATUS	
See Table 2-2.		Core
15:0	Current Performance State Value.	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:16	Reserved.	
Register Address: 199H, 409	IA32_PERF_CTL	
See Table 2-2.		Thread
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.		Thread
0	Reserved.	
3:1	On demand Clock Modulation Duty Cycle (R/W)	
4	On demand Clock Modulation Enable (R/W)	
63:5	Reserved.	
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) See Table 2-2.		Core
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Core
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		
0	Fast-Strings Enable See Table 2-2.	Thread
2:1	Reserved.	
3	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2. Default value is 1.	Thread
6:4	Reserved.	
7	Performance Monitoring Available (R) See Table 2-2.	Thread
10:8	Reserved.	
11	Branch Trace Storage Unavailable (R/O) See Table 2-2.	Thread
12	Processor Event Based Sampling Unavailable (R/O) See Table 2-2.	Thread
15:13	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.	Package
18	ENABLE MONITOR FSM. (R/W) See Table 2-2.	Thread
21:19	Reserved.	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	Bit Description	Scope
22	Limit CPUID Maxval (R/W) See Table 2-2.	Thread
23	xTPR Message Disable (R/W) See Table 2-2.	Thread
33:24	Reserved.	
34	XD Bit Disable (R/W) See Table 2-3.	Thread
37:35	Reserved.	
38	Turbo Mode Disable (R/W) When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.	Package
63:39	Reserved.	
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target		Thread
15:0	Reserved.	
23:16	Temperature Target (R) The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.	
63:24	Reserved.	
Register Address: 1A4H, 420	MSR_MISC_FEATURE_CONTROL	
Miscellaneous Feature Control (R/W)		
0	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.	Core
1	L2 Adjacent Cache Line Prefetcher Disable (R/W) If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).	Core
2	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.	Core
3	DCU IP Prefetcher Disable (R/W) If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.	Core
63:4	Reserved.	
Register Address: 1A6H, 422	MSR_OFFCORE_RSP_0	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Offcore Response Event Select Register (R/W)		Thread
Register Address: 1AAH, 426	MSR_MISC_PWR_MGMT	
Miscellaneous Power Management Control Various model specific features enumeration. See http://biosbits.org.		
0	EIST Hardware Coordination Disable (R/W) When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores. When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.	Package
1	Energy/Performance Bias Enable (R/W) This bit makes the IA32_ENERGY_PERF_BIAS register (MSR 1B0h) visible to software with Ring 0 privileges. This bit's status (1 or 0) is also reflected by CPUID.(EAX=06h):ECX[3].	Thread
63:2	Reserved.	
Register Address: 1ACH, 428	MSR_TURBO_POWER_CURRENT_LIMIT	
See http://biosbits.org.		
14:0	TDP Limit (R/W) TDP limit in 1/8 Watt granularity.	Package
15	TDP Limit Override Enable (R/W) A value = 0 indicates override is not active; a value = 1 indicates override is active.	Package
30:16	TDC Limit (R/W) TDC limit in 1/8 Amp granularity.	Package
31	TDC Limit Override Enable (R/W) A value = 0 indicates override is not active; a value = 1 indicates override is active.	Package
63:32	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
63:32	Reserved.	
Register Address: 1C8H, 456	MSR_LBR_SELECT	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record Filtering Select Register (R/W) See Section 19.9.2, "Filtering of Last Branch Records."		Core
0	CPL_EQ_0	
1	CPL_NEQ_0	
2	JCC	
3	NEAR_REL_CALL	
4	NEAR_IND_CALL	
5	NEAR_RET	
6	NEAR_IND_JMP	
7	NEAR_REL_JMP	
8	FAR_BRANCH	
63:9	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).		Thread
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W) See Table 2-2.		Thread
Register Address: 1DDH, 477	MSR_LER_FROM_LIP	
Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Thread
Register Address: 1DEH, 478	MSR_LER_TO_LIP	
Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Thread
Register Address: 1F2H, 498	IA32_SMRR_PHYSBASE	
See Table 2-2.		Core
Register Address: 1F3H, 499	IA32_SMRR_PHYSMASK	
See Table 2-2.		Core
Register Address: 1FCH, 508	MSR_POWER_CTL	
Power Control Register See http://biosbits.org.		Core
0	Reserved.	
1	C1E Enable (R/W) When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).	Package
63:2	Reserved.	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0	
See Table 2-2.		Thread
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0	
See Table 2-2.		Thread
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1	
See Table 2-2.		Thread
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1	
See Table 2-2.		Thread
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2	
See Table 2-2.		Thread
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2	
See Table 2-2.		Thread
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3	
See Table 2-2.		Thread
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3	
See Table 2-2.		Thread
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4	
See Table 2-2.		Thread
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4	
See Table 2-2.		Thread
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5	
See Table 2-2.		Thread
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5	
See Table 2-2.		Thread
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6	
See Table 2-2.		Thread
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6	
See Table 2-2.		Thread
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7	
See Table 2-2.		Thread
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7	
See Table 2-2.		Thread
Register Address: 210H, 528	IA32_MTRR_PHYSBASE8	
See Table 2-2.		Thread
Register Address: 211H, 529	IA32_MTRR_PHYSMASK8	
See Table 2-2.		Thread
Register Address: 212H, 530	IA32_MTRR_PHYSBASE9	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2.		Thread
Register Address: 213H, 531	IA32_MTRR_PHYSMASK9	
See Table 2-2.		Thread
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000	
See Table 2-2.		Thread
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000	
See Table 2-2.		Thread
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000	
See Table 2-2.		Thread
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000	
See Table 2-2.		Thread
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000	
See Table 2-2.		Thread
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000	
See Table 2-2.		Thread
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000	
See Table 2-2.		Thread
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000	
See Table 2-2.		Thread
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000	
See Table 2-2.		Thread
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000	
See Table 2-2.		Thread
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000	
See Table 2-2.		Thread
Register Address: 277H, 631	IA32_PAT	
See Table 2-2.		Thread
Register Address: 280H, 640	IA32_MC0_CTL2	
See Table 2-2.		Package
Register Address: 281H, 641	IA32_MC1_CTL2	
See Table 2-2.		Package
Register Address: 282H, 642	IA32_MC2_CTL2	
See Table 2-2.		Core
Register Address: 283H, 643	IA32_MC3_CTL2	
See Table 2-2.		Core
Register Address: 284H, 644	IA32_MC4_CTL2	
See Table 2-2.		Core

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 285H, 645	IA32_MC5_CTL2	
See Table 2-2.		Core
Register Address: 286H, 646	IA32_MC6_CTL2	
See Table 2-2.		Package
Register Address: 287H, 647	IA32_MC7_CTL2	
See Table 2-2.		Package
Register Address: 288H, 648	IA32_MC8_CTL2	
See Table 2-2.		Package
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE	
Default Memory Types (R/W) See Table 2-2.		Thread
Register Address: 309H, 777	IA32_FIXED_CTR0	
Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.		Thread
Register Address: 30AH, 778	IA32_FIXED_CTR1	
Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.		Thread
Register Address: 30BH, 779	IA32_FIXED_CTR2	
Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.		Thread
Register Address: 345H, 837	IA32_PERF_CAPABILITIES	
See Table 2-2. See Section 19.4.1, "IA32_DEBUGCTL MSR."		Thread
5:0	LBR Format See Table 2-2.	
6	PEBS Record Format	
7	PEBSSaveArchRegs See Table 2-2.	
11:8	PEBS_REC_FORMAT See Table 2-2.	
12	SMM_FREEZE See Table 2-2.	
63:13	Reserved.	
Register Address: 38DH, 909	IA32_FIXED_CTR_CTRL	
Fixed-Function-Counter Control Register (R/W) See Table 2-2.		Thread
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Thread
Register Address: 38EH, 910	MSR_PERF_GLOBAL_STATUS	
Provides single-bit status used by software to query the overflow condition of each performance counter. (R/O)		Thread

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
61	UNC_Ovf Uncore overflowed if 1.	
Register Address: 38FH, 911	IA32_PERF_GLOBAL_CTRL	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities."		Thread
Register Address: 390H, 912	IA32_PERF_GLOBAL_OVF_CTRL	
See Table 2-2. See Section 21.6.2.2, "Global Counter Control Facilities." Allows software to clear counter overflow conditions on any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.		Thread
Register Address: 390H, 912	MSR_PERF_GLOBAL_OVF_CTRL	
(R/W)		Thread
61	CLR_UNC_Ovf Set 1 to clear UNC_Ovf.	
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Section 21.3.1.1.1, "Processor Event Based Sampling (PEBS)."		Thread
0	Enable PEBS on IA32_PMC0 (R/W)	
1	Enable PEBS on IA32_PMC1 (R/W)	
2	Enable PEBS on IA32_PMC2 (R/W)	
3	Enable PEBS on IA32_PMC3 (R/W)	
31:4	Reserved.	
32	Enable Load Latency on IA32_PMC0 (R/W)	
33	Enable Load Latency on IA32_PMC1 (R/W)	
34	Enable Load Latency on IA32_PMC2 (R/W)	
35	Enable Load Latency on IA32_PMC3 (R/W)	
63:36	Reserved.	
Register Address: 3F6H, 1014	MSR_PEBS_LD_LAT	
See Section 21.3.1.1.2, "Load Latency Performance Monitoring Facility."		Thread
15:0	Minimum threshold latency value of tagged load operation that will be counted. (R/W)	
63:36	Reserved.	
Register Address: 3F8H, 1016	MSR_PKG_C3_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C3 Residency Counter (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.	
Register Address: 3F9H, 1017	MSR_PKG_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:0	Package C6 Residency Counter (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.	
Register Address: 3FAH, 1018	MSR_PKG_C7_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C7 Residency Counter (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.	
Register Address: 3FCH, 1020	MSR_CORE_C3_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C3 Residency Counter (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.	
Register Address: 3FDH, 1021	MSR_CORE_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.	
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Package
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Package
Register Address: 403H, 1027	IA32_MC0_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Package
Register Address: 406H, 1030	IA32_MC1_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Package

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 407H, 1031	IA32_MC1_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 408H, 1032	IA32_MC2_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Core
Register Address: 40BH, 1035	IA32_MC2_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 40CH, 1036	IA32_MC3_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 40DH, 1037	IA32_MC3_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 40EH, 1038	IA32_MC3_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Core
Register Address: 40FH, 1039	IA32_MC3_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 410H, 1040	IA32_MC4_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 411H, 1041	IA32_MC4_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 412H, 1042	IA32_MC4_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Core
Register Address: 413H, 1043	IA32_MC4_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 415H, 1045	IA32_MC5_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 417H, 1047	IA32_MC5_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 41BH, 1051	IA32_MC6_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 41CH, 1052	IA32_MC7_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 41DH, 1053	IA32_MC7_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 41EH, 1054	IA32_MC7_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 41FH, 1055	IA32_MC7_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 420H, 1056	IA32_MC8_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 421H, 1057	IA32_MC8_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 422H, 1058	IA32_MC8_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 423H, 1059	IA32_MC8_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 480H, 1152	IA32_VMX_BASIC	
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2 and Appendix A.1, "Basic VMX Information."		Thread
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	
Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2 and Appendix A.3, "VM-Execution Controls."		Thread
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Thread
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2 and Appendix A.4, “VM-Exit Controls.”		Thread
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2 and Appendix A.5, “VM-Entry Controls.”		Thread
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2 and Appendix A.6, “Miscellaneous Data.”		Thread
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2 and Appendix A.7, “VMX-Fixed Bits in CR0.”		Thread
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2 and Appendix A.7, “VMX-Fixed Bits in CR0.”		Thread
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2 and Appendix A.8, “VMX-Fixed Bits in CR4.”		Thread
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2 and Appendix A.8, “VMX-Fixed Bits in CR4.”		Thread
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	
Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2 and Appendix A.9, “VMCS Enumeration.”		Thread
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2	
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, “VM-Execution Controls.”		Thread
Register Address: 600H, 1536	IA32_DS_AREA	
DS Save Area (R/W) See Table 2-2 and Section 21.6.3.4, “Debug Store (DS) Mechanism.”		Thread
Register Address: 680H, 1664	MSR_LASTBRANCH_0_FROM_IP	
Last Branch Record 0 From IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ See Section 19.9.1 and record format in Section 19.4.8.1.		Thread
Register Address: 681H, 1665	MSR_LASTBRANCH_1_FROM_IP	
Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 682H, 1666	MSR_LASTBRANCH_2_FROM_IP	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 683H, 1667	MSR_LASTBRANCH_3_FROM_IP	
Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 684H, 1668	MSR_LASTBRANCH_4_FROM_IP	
Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 685H, 1669	MSR_LASTBRANCH_5_FROM_IP	
Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 686H, 1670	MSR_LASTBRANCH_6_FROM_IP	
Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 687H, 1671	MSR_LASTBRANCH_7_FROM_IP	
Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 688H, 1672	MSR_LASTBRANCH_8_FROM_IP	
Last Branch Record 8 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 689H, 1673	MSR_LASTBRANCH_9_FROM_IP	
Last Branch Record 9 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68AH, 1674	MSR_LASTBRANCH_10_FROM_IP	
Last Branch Record 10 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68BH, 1675	MSR_LASTBRANCH_11_FROM_IP	
Last Branch Record 11 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68CH, 1676	MSR_LASTBRANCH_12_FROM_IP	
Last Branch Record 12 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68DH, 1677	MSR_LASTBRANCH_13_FROM_IP	
Last Branch Record 13 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68EH, 1678	MSR_LASTBRANCH_14_FROM_IP	
Last Branch Record 14 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68FH, 1679	MSR_LASTBRANCH_15_FROM_IP	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 15 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 6C0H, 1728	MSR_LASTBRANCH_0_TO_IP	
Last Branch Record 0 To IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.		Thread
Register Address: 6C1H, 1729	MSR_LASTBRANCH_1_TO_IP	
Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C2H, 1730	MSR_LASTBRANCH_2_TO_IP	
Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C3H, 1731	MSR_LASTBRANCH_3_TO_IP	
Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C4H, 1732	MSR_LASTBRANCH_4_TO_IP	
Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C5H, 1733	MSR_LASTBRANCH_5_TO_IP	
Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C6H, 1734	MSR_LASTBRANCH_6_TO_IP	
Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C7H, 1735	MSR_LASTBRANCH_7_TO_IP	
Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C8H, 1736	MSR_LASTBRANCH_8_TO_IP	
Last Branch Record 8 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C9H, 1737	MSR_LASTBRANCH_9_TO_IP	
Last Branch Record 9 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CAH, 1738	MSR_LASTBRANCH_10_TO_IP	
Last Branch Record 10 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CBH, 1739	MSR_LASTBRANCH_11_TO_IP	
Last Branch Record 11 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 6CCH, 1740	MSR_LASTBRANCH_12_TO_IP	
Last Branch Record 12 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CDH, 1741	MSR_LASTBRANCH_13_TO_IP	
Last Branch Record 13 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CEH, 1742	MSR_LASTBRANCH_14_TO_IP	
Last Branch Record 14 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CFH, 1743	MSR_LASTBRANCH_15_TO_IP	
Last Branch Record 15 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 802H, 2050	IA32_X2APIC_APICID	
x2APIC ID Register (R/O)		Thread
Register Address: 803H, 2051	IA32_X2APIC_VERSION	
x2APIC Version Register (R/O)		Thread
Register Address: 808H, 2056	IA32_X2APIC_TPR	
x2APIC Task Priority Register (R/W)		Thread
Register Address: 80AH, 2058	IA32_X2APIC_PPR	
x2APIC Processor Priority Register (R/O)		Thread
Register Address: 80BH, 2059	IA32_X2APIC_EOI	
x2APIC EOI Register (W/O)		Thread
Register Address: 80DH, 2061	IA32_X2APIC_LDR	
x2APIC Logical Destination Register (R/O)		Thread
Register Address: 80FH, 2063	IA32_X2APIC_SIVR	
x2APIC Spurious Interrupt Vector Register (R/W)		Thread
Register Address: 810H, 2064	IA32_X2APIC_ISR0	
x2APIC In-Service Register Bits [31:0] (R/O)		Thread
Register Address: 811H, 2065	IA32_X2APIC_ISR1	
x2APIC In-Service Register Bits [63:32] (R/O)		Thread
Register Address: 812H, 2066	IA32_X2APIC_ISR2	
x2APIC In-Service Register Bits [95:64] (R/O)		Thread
Register Address: 813H, 2067	IA32_X2APIC_ISR3	
x2APIC In-Service Register Bits [127:96] (R/O)		Thread
Register Address: 814H, 2068	IA32_X2APIC_ISR4	
x2APIC In-Service Register Bits [159:128] (R/O)		Thread
Register Address: 815H, 2069	IA32_X2APIC_ISR5	
x2APIC In-Service Register Bits [191:160] (R/O)		Thread

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 816H, 2070	IA32_X2APIC_ISR6	
x2APIC In-Service Register Bits [223:192] (R/O)		Thread
Register Address: 817H, 2071	IA32_X2APIC_ISR7	
x2APIC In-Service Register Bits [255:224] (R/O)		Thread
Register Address: 818H, 2072	IA32_X2APIC_TMR0	
x2APIC Trigger Mode Register Bits [31:0] (R/O)		Thread
Register Address: 819H, 2073	IA32_X2APIC_TMR1	
x2APIC Trigger Mode Register Bits [63:32] (R/O)		Thread
Register Address: 81AH, 2074	IA32_X2APIC_TMR2	
x2APIC Trigger Mode Register Bits [95:64] (R/O)		Thread
Register Address: 81BH, 2075	IA32_X2APIC_TMR3	
x2APIC Trigger Mode Register Bits [127:96] (R/O)		Thread
Register Address: 81CH, 2076	IA32_X2APIC_TMR4	
x2APIC Trigger Mode Register Bits [159:128] (R/O)		Thread
Register Address: 81DH, 2077	IA32_X2APIC_TMR5	
x2APIC Trigger Mode Register Bits [191:160] (R/O)		Thread
Register Address: 81EH, 2078	IA32_X2APIC_TMR6	
x2APIC Trigger Mode Register Bits [223:192] (R/O)		Thread
Register Address: 81FH, 2079	IA32_X2APIC_TMR7	
x2APIC Trigger Mode Register Bits [255:224] (R/O)		Thread
Register Address: 820H, 2080	IA32_X2APIC_IRR0	
x2APIC Interrupt Request Register Bits [31:0] (R/O)		Thread
Register Address: 821H, 2081	IA32_X2APIC_IRR1	
x2APIC Interrupt Request Register Bits [63:32] (R/O)		Thread
Register Address: 822H, 2082	IA32_X2APIC_IRR2	
x2APIC Interrupt Request Register Bits [95:64] (R/O)		Thread
Register Address: 823H, 2083	IA32_X2APIC_IRR3	
x2APIC Interrupt Request Register Bits [127:96] (R/O)		Thread
Register Address: 824H, 2084	IA32_X2APIC_IRR4	
x2APIC Interrupt Request Register Bits [159:128] (R/O)		Thread
Register Address: 825H, 2085	IA32_X2APIC_IRR5	
x2APIC Interrupt Request Register Bits [191:160] (R/O)		Thread
Register Address: 826H, 2086	IA32_X2APIC_IRR6	
x2APIC Interrupt Request Register Bits [223:192] (R/O)		Thread
Register Address: 827H, 2087	IA32_X2APIC_IRR7	
x2APIC Interrupt Request Register Bits [255:224] (R/O)		Thread
Register Address: 828H, 2088	IA32_X2APIC_ESR	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
x2APIC Error Status Register (R/W)		Thread
Register Address: 82FH, 2095	IA32_X2APIC_LVT_CMCI	
x2APIC LVT Corrected Machine Check Interrupt Register (R/W)		Thread
Register Address: 830H, 2096	IA32_X2APIC_ICR	
x2APIC Interrupt Command Register (R/W)		Thread
Register Address: 832H, 2098	IA32_X2APIC_LVT_TIMER	
x2APIC LVT Timer Interrupt Register (R/W)		Thread
Register Address: 833H, 2099	IA32_X2APIC_LVT_THERMAL	
x2APIC LVT Thermal Sensor Interrupt Register (R/W)		Thread
Register Address: 834H, 2100	IA32_X2APIC_LVT_PMI	
x2APIC LVT Performance Monitor Register (R/W)		Thread
Register Address: 835H, 2101	IA32_X2APIC_LVT_LINT0	
x2APIC LVT LINT0 Register (R/W)		Thread
Register Address: 836H, 2102	IA32_X2APIC_LVT_LINT1	
x2APIC LVT LINT1 Register (R/W)		Thread
Register Address: 837H, 2103	IA32_X2APIC_LVT_ERROR	
x2APIC LVT Error Register (R/W)		Thread
Register Address: 838H, 2104	IA32_X2APIC_INIT_COUNT	
x2APIC Initial Count Register (R/W)		Thread
Register Address: 839H, 2105	IA32_X2APIC_CUR_COUNT	
x2APIC Current Count Register (R/O)		Thread
Register Address: 83EH, 2110	IA32_X2APIC_DIV_CONF	
x2APIC Divide Configuration Register (R/W)		Thread
Register Address: 83FH, 2111	IA32_X2APIC_SELF_IPI	
x2APIC Self IPI Register (W/O)		Thread
Register Address: C000_0080H	IA32_EFER	
Extended Feature Enables See Table 2-2.		Thread
Register Address: C000_0081H	IA32_STAR	
System Call Target Address (R/W) See Table 2-2.		Thread
Register Address: C000_0082H	IA32_LSTAR	
IA-32e Mode System Call Target Address (R/W) See Table 2-2.		Thread
Register Address: C000_0084H	IA32_FMASK	
System Call Flag Mask (R/W) See Table 2-2.		Thread
Register Address: C000_0100H	IA32_FS_BASE	

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Map of BASE Address of FS (R/W) See Table 2-2.		Thread
Register Address: C000_0101H	IA32_GS_BASE	
Map of BASE Address of GS (R/W) See Table 2-2.		Thread
Register Address: C000_0102H	IA32_KERNEL_GS_BASE	
Swap Target of BASE Address of GS (R/W) See Table 2-2.		Thread
Register Address: C000_0103H	IA32_TSC_AUX	
AUXILIARY TSC Signature (R/W) See Table 2-2 and Section 19.17.2, "IA32_TSC_AUX Register and RDTSCP Support."		Thread

2.8.1 Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

The Intel Xeon Processor 5500 and 3400 series supports additional model-specific registers listed in Table 2-16. These MSRs also apply to the Intel Core i7 and i5 processor family with a CPUID Signature DisplayFamily_DisplayModel value of 06_1AH, 06_1EH, or 06_1FH; see Table 2-1.

Table 2-16. Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Actual maximum turbo frequency is multiplied by 133.33MHz. (Not available in model 06_2EH.)		Package
7:0	Maximum Turbo Ratio Limit 1C (R/O) Maximum Turbo mode ratio limit with 1 core active.	
15:8	Maximum Turbo Ratio Limit 2C (R/O) Maximum Turbo mode ratio limit with 2 cores active.	
23:16	Maximum Turbo Ratio Limit 3C (R/O) Maximum Turbo mode ratio limit with 3 cores active.	
31:24	Maximum Turbo Ratio Limit 4C (R/O) Maximum Turbo mode ratio limit with 4 cores active.	
63:32	Reserved.	
Register Address: 301H, 769	MSR_GQ_SNOOP_MESF	
MSR_GQ_SNOOP_MESF		Package
0	From M to S (R/W)	
1	From E to S (R/W)	
2	From S to S (R/W)	
3	From F to S (R/W)	
4	From M to I (R/W)	

Table 2-16. Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
5	From E to I (R/W)	
6	From S to I (R/W)	
7	From F to I (R/W)	
63:8	Reserved.	
Register Address: 391H, 913	MSR_UNCORE_PERF_GLOBAL_CTRL	
See Section 21.3.1.2.1, "Uncore Performance Monitoring Management Facility."		Package
Register Address: 392H, 914	MSR_UNCORE_PERF_GLOBAL_STATUS	
See Section 21.3.1.2.1, "Uncore Performance Monitoring Management Facility."		Package
Register Address: 393H, 915	MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	
See Section 21.3.1.2.1, "Uncore Performance Monitoring Management Facility."		Package
Register Address: 394H, 916	MSR_UNCORE_FIXED_CTR0	
See Section 21.3.1.2.1, "Uncore Performance Monitoring Management Facility."		Package
Register Address: 395H, 917	MSR_UNCORE_FIXED_CTR_CTRL	
See Section 21.3.1.2.1, "Uncore Performance Monitoring Management Facility."		Package
Register Address: 396H, 918	MSR_UNCORE_ADDR_OPCODE_MATCH	
See Section 21.3.1.2.3, "Uncore Address/Opcode Match MSR."		Package
Register Address: 3B0H, 960	MSR_UNCORE_PMC0	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3B1H, 961	MSR_UNCORE_PMC1	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3B2H, 962	MSR_UNCORE_PMC2	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3B3H, 963	MSR_UNCORE_PMC3	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3B4H, 964	MSR_UNCORE_PMC4	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3B5H, 965	MSR_UNCORE_PMC5	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3B6H, 966	MSR_UNCORE_PMC6	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3B7H, 967	MSR_UNCORE_PMC7	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C0H, 944	MSR_UNCORE_PERFEVTSEL0	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C1H, 945	MSR_UNCORE_PERFEVTSEL1	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C2H, 946	MSR_UNCORE_PERFEVTSEL2	

Table 2-16. Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C3H, 947	MSR_UNCORE_PERFEVTSEL3	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C4H, 948	MSR_UNCORE_PERFEVTSEL4	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C5H, 949	MSR_UNCORE_PERFEVTSEL5	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C6H, 950	MSR_UNCORE_PERFEVTSEL6	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package
Register Address: 3C7H, 951	MSR_UNCORE_PERFEVTSEL7	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package

2.8.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

The Intel Xeon Processor 7500 series supports MSRs listed in Table 2-15 (except MSR address 1ADH) and additional model-specific registers listed in Table 2-17. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2EH.

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Reserved. Attempt to read/write will cause #UD.		Package
Register Address: 289H, 649	IA32_MC9_CTL2	
See Table 2-2.		Package
Register Address: 28AH, 650	IA32_MC10_CTL2	
See Table 2-2.		Package
Register Address: 28BH, 651	IA32_MC11_CTL2	
See Table 2-2.		Package
Register Address: 28CH, 652	IA32_MC12_CTL2	
See Table 2-2.		Package
Register Address: 28DH, 653	IA32_MC13_CTL2	
See Table 2-2.		Package
Register Address: 28EH, 654	IA32_MC14_CTL2	
See Table 2-2.		Package
Register Address: 28FH, 655	IA32_MC15_CTL2	
See Table 2-2.		Package
Register Address: 290H, 656	IA32_MC16_CTL2	
See Table 2-2.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 291H, 657	IA32_MC17_CTL2	
See Table 2-2.		Package
Register Address: 292H, 658	IA32_MC18_CTL2	
See Table 2-2.		Package
Register Address: 293H, 659	IA32_MC19_CTL2	
See Table 2-2.		Package
Register Address: 294H, 660	IA32_MC20_CTL2	
See Table 2-2.		Package
Register Address: 295H, 661	IA32_MC21_CTL2	
See Table 2-2.		Package
Register Address: 394H, 816	MSR_W_PMON_FIXED_CTR	
Uncore W-box PerfMon fixed counter.		Package
Register Address: 395H, 817	MSR_W_PMON_FIXED_CTR_CTL	
Uncore U-box PerfMon fixed counter control MSR.		Package
Register Address: 424H, 1060	IA32_MC9_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 425H, 1061	IA32_MC9_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 426H, 1062	IA32_MC9_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 427H, 1063	IA32_MC9_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 428H, 1064	IA32_MC10_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 429H, 1065	IA32_MC10_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 42AH, 1066	IA32_MC10_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 42BH, 1067	IA32_MC10_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 42CH, 1068	IA32_MC11_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 42DH, 1069	IA32_MC11_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 42EH, 1070	IA32_MC11_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 42FH, 1071	IA32_MC11_MISC	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 430H, 1072	IA32_MC12_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 431H, 1073	IA32_MC12_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 432H, 1074	IA32_MC12_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 433H, 1075	IA32_MC12_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 434H, 1076	IA32_MC13_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 435H, 1077	IA32_MC13_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 436H, 1078	IA32_MC13_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 437H, 1079	IA32_MC13_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 438H, 1080	IA32_MC14_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 439H, 1081	IA32_MC14_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 43AH, 1082	IA32_MC14_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 43BH, 1083	IA32_MC14_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 43CH, 1084	IA32_MC15_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 43DH, 1085	IA32_MC15_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 43EH, 1086	IA32_MC15_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 43FH, 1087	IA32_MC15_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 440H, 1088	IA32_MC16_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 441H, 1089	IA32_MC16_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 442H, 1090	IA32_MC16_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 443H, 1091	IA32_MC16_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 444H, 1092	IA32_MC17_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 445H, 1093	IA32_MC17_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 446H, 1094	IA32_MC17_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 447H, 1095	IA32_MC17_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 448H, 1096	IA32_MC18_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 449H, 1097	IA32_MC18_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 44AH, 1098	IA32_MC18_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 44BH, 1099	IA32_MC18_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 44CH, 1100	IA32_MC19_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 44DH, 1101	IA32_MC19_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 44EH, 1102	IA32_MC19_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 44FH, 1103	IA32_MC19_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 450H, 1104	IA32_MC20_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 451H, 1105	IA32_MC20_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 452H, 1106	IA32_MC20_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 453H, 1107	IA32_MC20_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 454H, 1108	IA32_MC21_CTL	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 455H, 1109	IA32_MC21_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 456H, 1110	IA32_MC21_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 457H, 1111	IA32_MC21_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: C00H, 3072	MSR_U_PMON_GLOBAL_CTRL	
Uncore U-box PerfMon global control MSR.		Package
Register Address: C01H, 3073	MSR_U_PMON_GLOBAL_STATUS	
Uncore U-box PerfMon global status MSR.		Package
Register Address: C02H, 3074	MSR_U_PMON_GLOBAL_OVF_CTRL	
Uncore U-box PerfMon global overflow control MSR.		Package
Register Address: C10H, 3088	MSR_U_PMON_EVNT_SEL	
Uncore U-box PerfMon event select MSR.		Package
Register Address: C11H, 3089	MSR_U_PMON_CTR	
Uncore U-box PerfMon counter MSR.		Package
Register Address: C20H, 3104	MSR_B0_PMON_BOX_CTRL	
Uncore B-box 0 PerfMon local box control MSR.		Package
Register Address: C21H, 3105	MSR_B0_PMON_BOX_STATUS	
Uncore B-box 0 PerfMon local box status MSR.		Package
Register Address: C22H, 3106	MSR_B0_PMON_BOX_OVF_CTRL	
Uncore B-box 0 PerfMon local box overflow control MSR.		Package
Register Address: C30H, 3120	MSR_B0_PMON_EVNT_SEL0	
Uncore B-box 0 PerfMon event select MSR.		Package
Register Address: C31H, 3121	MSR_B0_PMON_CTR0	
Uncore B-box 0 PerfMon counter MSR.		Package
Register Address: C32H, 3122	MSR_B0_PMON_EVNT_SEL1	
Uncore B-box 0 PerfMon event select MSR.		Package
Register Address: C33H, 3123	MSR_B0_PMON_CTR1	
Uncore B-box 0 PerfMon counter MSR.		Package
Register Address: C34H, 3124	MSR_B0_PMON_EVNT_SEL2	
Uncore B-box 0 PerfMon event select MSR.		Package
Register Address: C35H, 3125	MSR_B0_PMON_CTR2	
Uncore B-box 0 PerfMon counter MSR.		Package
Register Address: C36H, 3126	MSR_B0_PMON_EVNT_SEL3	
Uncore B-box 0 PerfMon event select MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: C37H, 3127	MSR_B0_PMON_CTR3	
Uncore B-box 0 PerfMon counter MSR.		Package
Register Address: C40H, 3136	MSR_S0_PMON_BOX_CTRL	
Uncore S-box 0 PerfMon local box control MSR.		Package
Register Address: C41H, 3137	MSR_S0_PMON_BOX_STATUS	
Uncore S-box 0 PerfMon local box status MSR.		Package
Register Address: C42H, 3138	MSR_S0_PMON_BOX_OVF_CTRL	
Uncore S-box 0 PerfMon local box overflow control MSR.		Package
Register Address: C50H, 3152	MSR_S0_PMON_EVNT_SEL0	
Uncore S-box 0 PerfMon event select MSR.		Package
Register Address: C51H, 3153	MSR_S0_PMON_CTR0	
Uncore S-box 0 PerfMon counter MSR.		Package
Register Address: C52H, 3154	MSR_S0_PMON_EVNT_SEL1	
Uncore S-box 0 PerfMon event select MSR.		Package
Register Address: C53H, 3155	MSR_S0_PMON_CTR1	
Uncore S-box 0 PerfMon counter MSR.		Package
Register Address: C54H, 3156	MSR_S0_PMON_EVNT_SEL2	
Uncore S-box 0 PerfMon event select MSR.		Package
Register Address: C55H, 3157	MSR_S0_PMON_CTR2	
Uncore S-box 0 PerfMon counter MSR.		Package
Register Address: C56H, 3158	MSR_S0_PMON_EVNT_SEL3	
Uncore S-box 0 PerfMon event select MSR.		Package
Register Address: C57H, 3159	MSR_S0_PMON_CTR3	
Uncore S-box 0 PerfMon counter MSR.		Package
Register Address: C60H, 3168	MSR_B1_PMON_BOX_CTRL	
Uncore B-box 1 PerfMon local box control MSR.		Package
Register Address: C61H, 3169	MSR_B1_PMON_BOX_STATUS	
Uncore B-box 1 PerfMon local box status MSR.		Package
Register Address: C62H, 3170	MSR_B1_PMON_BOX_OVF_CTRL	
Uncore B-box 1 PerfMon local box overflow control MSR.		Package
Register Address: C70H, 3184	MSR_B1_PMON_EVNT_SEL0	
Uncore B-box 1 PerfMon event select MSR.		Package
Register Address: C71H, 3185	MSR_B1_PMON_CTR0	
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C72H, 3186	MSR_B1_PMON_EVNT_SEL1	
Uncore B-box 1 PerfMon event select MSR.		Package
Register Address: C73H, 3187	MSR_B1_PMON_CTR1	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C74H, 3188	MSR_B1_PMON_EVNT_SEL2	
Uncore B-box 1 PerfMon event select MSR.		Package
Register Address: C75H, 3189	MSR_B1_PMON_CTR2	
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C76H, 3190	MSR_B1_PMON_EVNT_SEL3	
Uncore B-box 1vPerfMon event select MSR.		Package
Register Address: C77H, 3191	MSR_B1_PMON_CTR3	
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C80H, 3120	MSR_W_PMON_BOX_CTRL	
Uncore W-box PerfMon local box control MSR.		Package
Register Address: C81H, 3121	MSR_W_PMON_BOX_STATUS	
Uncore W-box PerfMon local box status MSR.		Package
Register Address: C82H, 3122	MSR_W_PMON_BOX_OVF_CTRL	
Uncore W-box PerfMon local box overflow control MSR.		Package
Register Address: C90H, 3136	MSR_W_PMON_EVNT_SEL0	
Uncore W-box PerfMon event select MSR.		Package
Register Address: C91H, 3137	MSR_W_PMON_CTR0	
Uncore W-box PerfMon counter MSR.		Package
Register Address: C92H, 3138	MSR_W_PMON_EVNT_SEL1	
Uncore W-box PerfMon event select MSR.		Package
Register Address: C93H, 3139	MSR_W_PMON_CTR1	
Uncore W-box PerfMon counter MSR.		Package
Register Address: C94H, 3140	MSR_W_PMON_EVNT_SEL2	
Uncore W-box PerfMon event select MSR.		Package
Register Address: C95H, 3141	MSR_W_PMON_CTR2	
Uncore W-box PerfMon counter MSR.		Package
Register Address: C96H, 3142	MSR_W_PMON_EVNT_SEL3	
Uncore W-box PerfMon event select MSR.		Package
Register Address: C97H, 3143	MSR_W_PMON_CTR3	
Uncore W-box PerfMon counter MSR.		Package
Register Address: CA0H, 3232	MSR_M0_PMON_BOX_CTRL	
Uncore M-box 0 PerfMon local box control MSR.		Package
Register Address: CA1H, 3233	MSR_M0_PMON_BOX_STATUS	
Uncore M-box 0 PerfMon local box status MSR.		Package
Register Address: CA2H, 3234	MSR_M0_PMON_BOX_OVF_CTRL	
Uncore M-box 0 PerfMon local box overflow control MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: CA4H, 3236	MSR_M0_PMON_TIMESTAMP	
Uncore M-box 0 PerfMon time stamp unit select MSR.		Package
Register Address: CA5H, 3237	MSR_M0_PMON_DSP	
Uncore M-box 0 PerfMon DSP unit select MSR.		Package
Register Address: CA6H, 3238	MSR_M0_PMON_ISS	
Uncore M-box 0 PerfMon ISS unit select MSR.		Package
Register Address: CA7H, 3239	MSR_M0_PMON_MAP	
Uncore M-box 0 PerfMon MAP unit select MSR.		Package
Register Address: CA8H, 3240	MSR_M0_PMON_MSC_THR	
Uncore M-box 0 PerfMon MIC THR select MSR.		Package
Register Address: CA9H, 3241	MSR_M0_PMON_PGT	
Uncore M-box 0 PerfMon PGT unit select MSR.		Package
Register Address: CAAH, 3242	MSR_M0_PMON_PLD	
Uncore M-box 0 PerfMon PLD unit select MSR.		Package
Register Address: CABH, 3243	MSR_M0_PMON_ZDP	
Uncore M-box 0 PerfMon ZDP unit select MSR.		Package
Register Address: CB0H, 3248	MSR_M0_PMON_EVNT_SEL0	
Uncore M-box 0 PerfMon event select MSR.		Package
Register Address: CB1H, 3249	MSR_M0_PMON_CTR0	
Uncore M-box 0 PerfMon counter MSR.		Package
Register Address: CB2H, 3250	MSR_M0_PMON_EVNT_SEL1	
Uncore M-box 0 PerfMon event select MSR.		Package
Register Address: CB3H, 3251	MSR_M0_PMON_CTR1	
Uncore M-box 0 PerfMon counter MSR.		Package
Register Address: CB4H, 3252	MSR_M0_PMON_EVNT_SEL2	
Uncore M-box 0 PerfMon event select MSR.		Package
Register Address: CB5H, 3253	MSR_M0_PMON_CTR2	
Uncore M-box 0 PerfMon counter MSR.		Package
Register Address: CB6H, 3254	MSR_M0_PMON_EVNT_SEL3	
Uncore M-box 0 PerfMon event select MSR.		Package
Register Address: CB7H, 3255	MSR_M0_PMON_CTR3	
Uncore M-box 0 PerfMon counter MSR.		Package
Register Address: CB8H, 3256	MSR_M0_PMON_EVNT_SEL4	
Uncore M-box 0 PerfMon event select MSR.		Package
Register Address: CB9H, 3257	MSR_M0_PMON_CTR4	
Uncore M-box 0 PerfMon counter MSR.		Package
Register Address: CBAH, 3258	MSR_M0_PMON_EVNT_SEL5	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore M-box 0 PerfMon event select MSR.		Package
Register Address: CBBH, 3259	MSR_M0_PMON_CTR5	
Uncore M-box 0 PerfMon counter MSR.		Package
Register Address: CC0H, 3264	MSR_S1_PMON_BOX_CTRL	
Uncore S-box 1 PerfMon local box control MSR.		Package
Register Address: CC1H, 3265	MSR_S1_PMON_BOX_STATUS	
Uncore S-box 1 PerfMon local box status MSR.		Package
Register Address: CC2H, 3266	MSR_S1_PMON_BOX_OVF_CTRL	
Uncore S-box 1 PerfMon local box overflow control MSR.		Package
Register Address: CD0H, 3280	MSR_S1_PMON_EVNT_SEL0	
Uncore S-box 1 PerfMon event select MSR.		Package
Register Address: CD1H, 3281	MSR_S1_PMON_CTR0	
Uncore S-box 1 PerfMon counter MSR.		Package
Register Address: CD2H, 3282	MSR_S1_PMON_EVNT_SEL1	
Uncore S-box 1 PerfMon event select MSR.		Package
Register Address: CD3H, 3283	MSR_S1_PMON_CTR1	
Uncore S-box 1 PerfMon counter MSR.		Package
Register Address: CD4H, 3284	MSR_S1_PMON_EVNT_SEL2	
Uncore S-box 1 PerfMon event select MSR.		Package
Register Address: CD5H, 3285	MSR_S1_PMON_CTR2	
Uncore S-box 1 PerfMon counter MSR.		Package
Register Address: CD6H, 3286	MSR_S1_PMON_EVNT_SEL3	
Uncore S-box 1 PerfMon event select MSR.		Package
Register Address: CD7H, 3287	MSR_S1_PMON_CTR3	
Uncore S-box 1 PerfMon counter MSR.		Package
Register Address: CE0H, 3296	MSR_M1_PMON_BOX_CTRL	
Uncore M-box 1 PerfMon local box control MSR.		Package
Register Address: CE1H, 3297	MSR_M1_PMON_BOX_STATUS	
Uncore M-box 1 PerfMon local box status MSR.		Package
Register Address: CE2H, 3298	MSR_M1_PMON_BOX_OVF_CTRL	
Uncore M-box 1 PerfMon local box overflow control MSR.		Package
Register Address: CE4H, 3300	MSR_M1_PMON_TIMESTAMP	
Uncore M-box 1 PerfMon time stamp unit select MSR.		Package
Register Address: CE5H, 3301	MSR_M1_PMON_DSP	
Uncore M-box 1 PerfMon DSP unit select MSR.		Package
Register Address: CE6H, 3302	MSR_M1_PMON_ISS	
Uncore M-box 1 PerfMon ISS unit select MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: CE7H, 3303	MSR_M1_PMON_MAP	
Uncore M-box 1 PerfMon MAP unit select MSR.		Package
Register Address: CE8H, 3304	MSR_M1_PMON_MSC_THR	
Uncore M-box 1 PerfMon MIC THR select MSR.		Package
Register Address: CE9H, 3305	MSR_M1_PMON_PGT	
Uncore M-box 1 PerfMon PGT unit select MSR.		Package
Register Address: CEAH, 3306	MSR_M1_PMON_PLD	
Uncore M-box 1 PerfMon PLD unit select MSR.		Package
Register Address: CEBH, 3307	MSR_M1_PMON_ZDP	
Uncore M-box 1 PerfMon ZDP unit select MSR.		Package
Register Address: CF0H, 3312	MSR_M1_PMON_EVNT_SEL0	
Uncore M-box 1 PerfMon event select MSR.		Package
Register Address: CF1H, 3313	MSR_M1_PMON_CTR0	
Uncore M-box 1 PerfMon counter MSR.		Package
Register Address: CF2H, 3314	MSR_M1_PMON_EVNT_SEL1	
Uncore M-box 1 PerfMon event select MSR.		Package
Register Address: CF3H, 3315	MSR_M1_PMON_CTR1	
Uncore M-box 1 PerfMon counter MSR.		Package
Register Address: CF4H, 3316	MSR_M1_PMON_EVNT_SEL2	
Uncore M-box 1 PerfMon event select MSR.		Package
Register Address: CF5H, 3317	MSR_M1_PMON_CTR2	
Uncore M-box 1 PerfMon counter MSR.		Package
Register Address: CF6H, 3318	MSR_M1_PMON_EVNT_SEL3	
Uncore M-box 1 PerfMon event select MSR.		Package
Register Address: CF7H, 3319	MSR_M1_PMON_CTR3	
Uncore M-box 1 PerfMon counter MSR.		Package
Register Address: CF8H, 3320	MSR_M1_PMON_EVNT_SEL4	
Uncore M-box 1 PerfMon event select MSR.		Package
Register Address: CF9H, 3321	MSR_M1_PMON_CTR4	
Uncore M-box 1 PerfMon counter MSR.		Package
Register Address: CFAH, 3322	MSR_M1_PMON_EVNT_SEL5	
Uncore M-box 1 PerfMon event select MSR.		Package
Register Address: CFBH, 3323	MSR_M1_PMON_CTR5	
Uncore M-box 1 PerfMon counter MSR.		Package
Register Address: D00H, 3328	MSR_C0_PMON_BOX_CTRL	
Uncore C-box 0 PerfMon local box control MSR.		Package
Register Address: D01H, 3329	MSR_C0_PMON_BOX_STATUS	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-box 0 PerfMon local box status MSR.		Package
Register Address: D02H, 3330	MSR_C0_PMON_BOX_OVF_CTRL	
Uncore C-box 0 PerfMon local box overflow control MSR.		Package
Register Address: D10H, 3344	MSR_C0_PMON_EVNT_SEL0	
Uncore C-box 0 PerfMon event select MSR.		Package
Register Address: D11H, 3345	MSR_C0_PMON_CTR0	
Uncore C-box 0 PerfMon counter MSR.		Package
Register Address: D12H, 3346	MSR_C0_PMON_EVNT_SEL1	
Uncore C-box 0 PerfMon event select MSR.		Package
Register Address: D13H, 3347	MSR_C0_PMON_CTR1	
Uncore C-box 0 PerfMon counter MSR.		Package
Register Address: D14H, 3348	MSR_C0_PMON_EVNT_SEL2	
Uncore C-box 0 PerfMon event select MSR.		Package
Register Address: D15H, 3349	MSR_C0_PMON_CTR2	
Uncore C-box 0 PerfMon counter MSR.		Package
Register Address: D16H, 3350	MSR_C0_PMON_EVNT_SEL3	
Uncore C-box 0 PerfMon event select MSR.		Package
Register Address: D17H, 3351	MSR_C0_PMON_CTR3	
Uncore C-box 0 PerfMon counter MSR.		Package
Register Address: D18H, 3352	MSR_C0_PMON_EVNT_SEL4	
Uncore C-box 0 PerfMon event select MSR.		Package
Register Address: D19H, 3353	MSR_C0_PMON_CTR4	
Uncore C-box 0 PerfMon counter MSR.		Package
Register Address: D1AH, 3354	MSR_C0_PMON_EVNT_SEL5	
Uncore C-box 0 PerfMon event select MSR.		Package
Register Address: D1BH, 3355	MSR_C0_PMON_CTR5	
Uncore C-box 0 PerfMon counter MSR.		Package
Register Address: D20H, 3360	MSR_C4_PMON_BOX_CTRL	
Uncore C-box 4 PerfMon local box control MSR.		Package
Register Address: D21H, 3361	MSR_C4_PMON_BOX_STATUS	
Uncore C-box 4 PerfMon local box status MSR.		Package
Register Address: D22H, 3362	MSR_C4_PMON_BOX_OVF_CTRL	
Uncore C-box 4 PerfMon local box overflow control MSR.		Package
Register Address: D30H, 3376	MSR_C4_PMON_EVNT_SEL0	
Uncore C-box 4 PerfMon event select MSR.		Package
Register Address: D31H, 3377	MSR_C4_PMON_CTR0	
Uncore C-box 4 PerfMon counter MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: D32H, 3378	MSR_C4_PMON_EVNT_SEL1	
Uncore C-box 4 PerfMon event select MSR.		Package
Register Address: D33H, 3379	MSR_C4_PMON_CTR1	
Uncore C-box 4 PerfMon counter MSR.		Package
Register Address: D34H, 3380	MSR_C4_PMON_EVNT_SEL2	
Uncore C-box 4 PerfMon event select MSR.		Package
Register Address: D35H, 3381	MSR_C4_PMON_CTR2	
Uncore C-box 4 PerfMon counter MSR.		Package
Register Address: D36H, 3382	MSR_C4_PMON_EVNT_SEL3	
Uncore C-box 4 PerfMon event select MSR.		Package
Register Address: D37H, 3383	MSR_C4_PMON_CTR3	
Uncore C-box 4 PerfMon counter MSR.		Package
Register Address: D38H, 3384	MSR_C4_PMON_EVNT_SEL4	
Uncore C-box 4 PerfMon event select MSR.		Package
Register Address: D39H, 3385	MSR_C4_PMON_CTR4	
Uncore C-box 4 PerfMon counter MSR.		Package
Register Address: D3AH, 3386	MSR_C4_PMON_EVNT_SEL5	
Uncore C-box 4 PerfMon event select MSR.		Package
Register Address: D3BH, 3387	MSR_C4_PMON_CTR5	
Uncore C-box 4 PerfMon counter MSR.		Package
Register Address: D40H, 3392	MSR_C2_PMON_BOX_CTRL	
Uncore C-box 2 PerfMon local box control MSR.		Package
Register Address: D41H, 3393	MSR_C2_PMON_BOX_STATUS	
Uncore C-box 2 PerfMon local box status MSR.		Package
Register Address: D42H, 3394	MSR_C2_PMON_BOX_OVF_CTRL	
Uncore C-box 2 PerfMon local box overflow control MSR.		Package
Register Address: D50H, 3408	MSR_C2_PMON_EVNT_SEL0	
Uncore C-box 2 PerfMon event select MSR.		Package
Register Address: D51H, 3409	MSR_C2_PMON_CTR0	
Uncore C-box 2 PerfMon counter MSR.		Package
Register Address: D52H, 3410	MSR_C2_PMON_EVNT_SEL1	
Uncore C-box 2 PerfMon event select MSR.		Package
Register Address: D53H, 3411	MSR_C2_PMON_CTR1	
Uncore C-box 2 PerfMon counter MSR.		Package
Register Address: D54H, 3412	MSR_C2_PMON_EVNT_SEL2	
Uncore C-box 2 PerfMon event select MSR.		Package
Register Address: D55H, 3413	MSR_C2_PMON_CTR2	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-box 2 PerfMon counter MSR.		Package
Register Address: D56H, 3414	MSR_C2_PMON_EVNT_SEL3	
Uncore C-box 2 PerfMon event select MSR.		Package
Register Address: D57H, 3415	MSR_C2_PMON_CTR3	
Uncore C-box 2 PerfMon counter MSR.		Package
Register Address: D58H, 3416	MSR_C2_PMON_EVNT_SEL4	
Uncore C-box 2 PerfMon event select MSR.		Package
Register Address: D59H, 3417	MSR_C2_PMON_CTR4	
Uncore C-box 2 PerfMon counter MSR.		Package
Register Address: D5AH, 3418	MSR_C2_PMON_EVNT_SEL5	
Uncore C-box 2 PerfMon event select MSR.		Package
Register Address: D5BH, 3419	MSR_C2_PMON_CTR5	
Uncore C-box 2 PerfMon counter MSR.		Package
Register Address: D60H, 3424	MSR_C6_PMON_BOX_CTRL	
Uncore C-box 6 PerfMon local box control MSR.		Package
Register Address: D61H, 3425	MSR_C6_PMON_BOX_STATUS	
Uncore C-box 6 PerfMon local box status MSR.		Package
Register Address: D62H, 3426	MSR_C6_PMON_BOX_OVF_CTRL	
Uncore C-box 6 PerfMon local box overflow control MSR.		Package
Register Address: D70H, 3440	MSR_C6_PMON_EVNT_SEL0	
Uncore C-box 6 PerfMon event select MSR.		Package
Register Address: D71H, 3441	MSR_C6_PMON_CTR0	
Uncore C-box 6 PerfMon counter MSR.		Package
Register Address: D72H, 3442	MSR_C6_PMON_EVNT_SEL1	
Uncore C-box 6 PerfMon event select MSR.		Package
Register Address: D73H, 3443	MSR_C6_PMON_CTR1	
Uncore C-box 6 PerfMon counter MSR.		Package
Register Address: D74H, 3444	MSR_C6_PMON_EVNT_SEL2	
Uncore C-box 6 PerfMon event select MSR.		Package
Register Address: D75H, 3445	MSR_C6_PMON_CTR2	
Uncore C-box 6 PerfMon counter MSR.		Package
Register Address: D76H, 3446	MSR_C6_PMON_EVNT_SEL3	
Uncore C-box 6 PerfMon event select MSR.		Package
Register Address: D77H, 3447	MSR_C6_PMON_CTR3	
Uncore C-box 6 PerfMon counter MSR.		Package
Register Address: D78H, 3448	MSR_C6_PMON_EVNT_SEL4	
Uncore C-box 6 PerfMon event select MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: D79H, 3449	MSR_C6_PMON_CTR4	
Uncore C-box 6 PerfMon counter MSR.		Package
Register Address: D7AH, 3450	MSR_C6_PMON_EVNT_SEL5	
Uncore C-box 6 PerfMon event select MSR.		Package
Register Address: D7BH, 3451	MSR_C6_PMON_CTR5	
Uncore C-box 6 PerfMon counter MSR.		Package
Register Address: D80H, 3456	MSR_C1_PMON_BOX_CTRL	
Uncore C-box 1 PerfMon local box control MSR.		Package
Register Address: D81H, 3457	MSR_C1_PMON_BOX_STATUS	
Uncore C-box 1 PerfMon local box status MSR.		Package
Register Address: D82H, 3458	MSR_C1_PMON_BOX_OVF_CTRL	
Uncore C-box 1 PerfMon local box overflow control MSR.		Package
Register Address: D90H, 3472	MSR_C1_PMON_EVNT_SEL0	
Uncore C-box 1 PerfMon event select MSR.		Package
Register Address: D91H, 3473	MSR_C1_PMON_CTR0	
Uncore C-box 1 PerfMon counter MSR.		Package
Register Address: D92H, 3474	MSR_C1_PMON_EVNT_SEL1	
Uncore C-box 1 PerfMon event select MSR.		Package
Register Address: D93H, 3475	MSR_C1_PMON_CTR1	
Uncore C-box 1 PerfMon counter MSR.		Package
Register Address: D94H, 3476	MSR_C1_PMON_EVNT_SEL2	
Uncore C-box 1 PerfMon event select MSR.		Package
Register Address: D95H, 3477	MSR_C1_PMON_CTR2	
Uncore C-box 1 PerfMon counter MSR.		Package
Register Address: D96H, 3478	MSR_C1_PMON_EVNT_SEL3	
Uncore C-box 1 PerfMon event select MSR.		Package
Register Address: D97H, 3479	MSR_C1_PMON_CTR3	
Uncore C-box 1 PerfMon counter MSR.		Package
Register Address: D98H, 3480	MSR_C1_PMON_EVNT_SEL4	
Uncore C-box 1 PerfMon event select MSR.		Package
Register Address: D99H, 3481	MSR_C1_PMON_CTR4	
Uncore C-box 1 PerfMon counter MSR.		Package
Register Address: D9AH, 3482	MSR_C1_PMON_EVNT_SEL5	
Uncore C-box 1 PerfMon event select MSR.		Package
Register Address: D9BH, 3483	MSR_C1_PMON_CTR5	
Uncore C-box 1 PerfMon counter MSR.		Package
Register Address: DA0H, 3488	MSR_C5_PMON_BOX_CTRL	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-box 5 PerfMon local box control MSR.		Package
Register Address: DA1H, 3489	MSR_C5_PMON_BOX_STATUS	
Uncore C-box 5 PerfMon local box status MSR.		Package
Register Address: DA2H, 3490	MSR_C5_PMON_BOX_OVF_CTRL	
Uncore C-box 5 PerfMon local box overflow control MSR.		Package
Register Address: DB0H, 3504	MSR_C5_PMON_EVNT_SEL0	
Uncore C-box 5 PerfMon event select MSR.		Package
Register Address: DB1H, 3505	MSR_C5_PMON_CTR0	
Uncore C-box 5 PerfMon counter MSR.		Package
Register Address: DB2H, 3506	MSR_C5_PMON_EVNT_SEL1	
Uncore C-box 5 PerfMon event select MSR.		Package
Register Address: DB3H, 3507	MSR_C5_PMON_CTR1	
Uncore C-box 5 PerfMon counter MSR.		Package
Register Address: DB4H, 3508	MSR_C5_PMON_EVNT_SEL2	
Uncore C-box 5 PerfMon event select MSR.		Package
Register Address: DB5H, 3509	MSR_C5_PMON_CTR2	
Uncore C-box 5 PerfMon counter MSR.		Package
Register Address: DB6H, 3510	MSR_C5_PMON_EVNT_SEL3	
Uncore C-box 5 PerfMon event select MSR.		Package
Register Address: DB7H, 3511	MSR_C5_PMON_CTR3	
Uncore C-box 5 PerfMon counter MSR.		Package
Register Address: DB8H, 3512	MSR_C5_PMON_EVNT_SEL4	
Uncore C-box 5 PerfMon event select MSR.		Package
Register Address: DB9H, 3513	MSR_C5_PMON_CTR4	
Uncore C-box 5 PerfMon counter MSR.		Package
Register Address: DBAH, 3514	MSR_C5_PMON_EVNT_SEL5	
Uncore C-box 5 PerfMon event select MSR.		Package
Register Address: DBBH, 3515	MSR_C5_PMON_CTR5	
Uncore C-box 5 PerfMon counter MSR.		Package
Register Address: DC0H, 3520	MSR_C3_PMON_BOX_CTRL	
Uncore C-box 3 PerfMon local box control MSR.		Package
Register Address: DC1H, 3521	MSR_C3_PMON_BOX_STATUS	
Uncore C-box 3 PerfMon local box status MSR.		Package
Register Address: DC2H, 3522	MSR_C3_PMON_BOX_OVF_CTRL	
Uncore C-box 3 PerfMon local box overflow control MSR.		Package
Register Address: DD0H, 3536	MSR_C3_PMON_EVNT_SEL0	
Uncore C-box 3 PerfMon event select MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: DD1H, 3537	MSR_C3_PMON_CTR0	
Uncore C-box 3 PerfMon counter MSR.		Package
Register Address: DD2H, 3538	MSR_C3_PMON_EVNT_SEL1	
Uncore C-box 3 PerfMon event select MSR.		Package
Register Address: DD3H, 3539	MSR_C3_PMON_CTR1	
Uncore C-box 3 PerfMon counter MSR.		Package
Register Address: DD4H, 3540	MSR_C3_PMON_EVNT_SEL2	
Uncore C-box 3 PerfMon event select MSR.		Package
Register Address: DD5H, 3541	MSR_C3_PMON_CTR2	
Uncore C-box 3 PerfMon counter MSR.		Package
Register Address: DD6H, 3542	MSR_C3_PMON_EVNT_SEL3	
Uncore C-box 3 PerfMon event select MSR.		Package
Register Address: DD7H, 3543	MSR_C3_PMON_CTR3	
Uncore C-box 3 PerfMon counter MSR.		Package
Register Address: DD8H, 3544	MSR_C3_PMON_EVNT_SEL4	
Uncore C-box 3 PerfMon event select MSR.		Package
Register Address: DD9H, 3545	MSR_C3_PMON_CTR4	
Uncore C-box 3 PerfMon counter MSR.		Package
Register Address: DDAH, 3546	MSR_C3_PMON_EVNT_SEL5	
Uncore C-box 3 PerfMon event select MSR.		Package
Register Address: DDBH, 3547	MSR_C3_PMON_CTR5	
Uncore C-box 3 PerfMon counter MSR.		Package
Register Address: DE0H, 3552	MSR_C7_PMON_BOX_CTRL	
Uncore C-box 7 PerfMon local box control MSR.		Package
Register Address: DE1H, 3553	MSR_C7_PMON_BOX_STATUS	
Uncore C-box 7 PerfMon local box status MSR.		Package
Register Address: DE2H, 3554	MSR_C7_PMON_BOX_OVF_CTRL	
Uncore C-box 7 PerfMon local box overflow control MSR.		Package
Register Address: DF0H, 3568	MSR_C7_PMON_EVNT_SEL0	
Uncore C-box 7 PerfMon event select MSR.		Package
Register Address: DF1H, 3569	MSR_C7_PMON_CTR0	
Uncore C-box 7 PerfMon counter MSR.		Package
Register Address: DF2H, 3570	MSR_C7_PMON_EVNT_SEL1	
Uncore C-box 7 PerfMon event select MSR.		Package
Register Address: DF3H, 3571	MSR_C7_PMON_CTR1	
Uncore C-box 7 PerfMon counter MSR.		Package
Register Address: DF4H, 3572	MSR_C7_PMON_EVNT_SEL2	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-box 7 PerfMon event select MSR.		Package
Register Address: DF5H, 3573	MSR_C7_PMON_CTR2	
Uncore C-box 7 PerfMon counter MSR.		Package
Register Address: DF6H, 3574	MSR_C7_PMON_EVNT_SEL3	
Uncore C-box 7 PerfMon event select MSR.		Package
Register Address: DF7H, 3575	MSR_C7_PMON_CTR3	
Uncore C-box 7 PerfMon counter MSR.		Package
Register Address: DF8H, 3576	MSR_C7_PMON_EVNT_SEL4	
Uncore C-box 7 PerfMon event select MSR.		Package
Register Address: DF9H, 3577	MSR_C7_PMON_CTR4	
Uncore C-box 7 PerfMon counter MSR.		Package
Register Address: DFAH, 3578	MSR_C7_PMON_EVNT_SEL5	
Uncore C-box 7 PerfMon event select MSR.		Package
Register Address: DFBH, 3579	MSR_C7_PMON_CTR5	
Uncore C-box 7 PerfMon counter MSR.		Package
Register Address: E00H, 3584	MSR_R0_PMON_BOX_CTRL	
Uncore R-box 0 PerfMon local box control MSR.		Package
Register Address: E01H, 3585	MSR_R0_PMON_BOX_STATUS	
Uncore R-box 0 PerfMon local box status MSR.		Package
Register Address: E02H, 3586	MSR_R0_PMON_BOX_OVF_CTRL	
Uncore R-box 0 PerfMon local box overflow control MSR.		Package
Register Address: E04H, 3588	MSR_R0_PMON_IPERF0_P0	
Uncore R-box 0 PerfMon IPERF0 unit Port 0 select MSR.		Package
Register Address: E05H, 3589	MSR_R0_PMON_IPERF0_P1	
Uncore R-box 0 PerfMon IPERF0 unit Port 1 select MSR.		Package
Register Address: E06H, 3590	MSR_R0_PMON_IPERF0_P2	
Uncore R-box 0 PerfMon IPERF0 unit Port 2 select MSR.		Package
Register Address: E07H, 3591	MSR_R0_PMON_IPERF0_P3	
Uncore R-box 0 PerfMon IPERF0 unit Port 3 select MSR.		Package
Register Address: E08H, 3592	MSR_R0_PMON_IPERF0_P4	
Uncore R-box 0 PerfMon IPERF0 unit Port 4 select MSR.		Package
Register Address: E09H, 3593	MSR_R0_PMON_IPERF0_P5	
Uncore R-box 0 PerfMon IPERF0 unit Port 5 select MSR.		Package
Register Address: E0AH, 3594	MSR_R0_PMON_IPERF0_P6	
Uncore R-box 0 PerfMon IPERF0 unit Port 6 select MSR.		Package
Register Address: E0BH, 3595	MSR_R0_PMON_IPERF0_P7	
Uncore R-box 0 PerfMon IPERF0 unit Port 7 select MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E0CH, 3596	MSR_R0_PMON_QLX_P0	
Uncore R-box 0 PerfMon QLX unit Port 0 select MSR.		Package
Register Address: E0DH, 3597	MSR_R0_PMON_QLX_P1	
Uncore R-box 0 PerfMon QLX unit Port 1 select MSR.		Package
Register Address: E0EH, 3598	MSR_R0_PMON_QLX_P2	
Uncore R-box 0 PerfMon QLX unit Port 2 select MSR.		Package
Register Address: E0FH, 3599	MSR_R0_PMON_QLX_P3	
Uncore R-box 0 PerfMon QLX unit Port 3 select MSR.		Package
Register Address: E10H, 3600	MSR_R0_PMON_EVNT_SEL0	
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E11H, 3601	MSR_R0_PMON_CTR0	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E12H, 3602	MSR_R0_PMON_EVNT_SEL1	
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E13H, 3603	MSR_R0_PMON_CTR1	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E14H, 3604	MSR_R0_PMON_EVNT_SEL2	
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E15H, 3605	MSR_R0_PMON_CTR2	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E16H, 3606	MSR_R0_PMON_EVNT_SEL3	
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E17H, 3607	MSR_R0_PMON_CTR3	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E18H, 3608	MSR_R0_PMON_EVNT_SEL4	
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E19H, 3609	MSR_R0_PMON_CTR4	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E1AH, 3610	MSR_R0_PMON_EVNT_SEL5	
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E1BH, 3611	MSR_R0_PMON_CTR5	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E1CH, 3612	MSR_R0_PMON_EVNT_SEL6	
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E1DH, 3613	MSR_R0_PMON_CTR6	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E1EH, 3614	MSR_R0_PMON_EVNT_SEL7	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore R-box 0 PerfMon event select MSR.		Package
Register Address: E1FH, 3615	MSR_R0_PMON_CTR7	
Uncore R-box 0 PerfMon counter MSR.		Package
Register Address: E20H, 3616	MSR_R1_PMON_BOX_CTRL	
Uncore R-box 1 PerfMon local box control MSR.		Package
Register Address: E21H, 3617	MSR_R1_PMON_BOX_STATUS	
Uncore R-box 1 PerfMon local box status MSR.		Package
Register Address: E22H, 3618	MSR_R1_PMON_BOX_OVF_CTRL	
Uncore R-box 1 PerfMon local box overflow control MSR.		Package
Register Address: E24H, 3620	MSR_R1_PMON_IPERF1_P8	
Uncore R-box 1 PerfMon IPERF1 unit Port 8 select MSR.		Package
Register Address: E25H, 3621	MSR_R1_PMON_IPERF1_P9	
Uncore R-box 1 PerfMon IPERF1 unit Port 9 select MSR.		Package
Register Address: E26H, 3622	MSR_R1_PMON_IPERF1_P10	
Uncore R-box 1 PerfMon IPERF1 unit Port 10 select MSR.		Package
Register Address: E27H, 3623	MSR_R1_PMON_IPERF1_P11	
Uncore R-box 1 PerfMon IPERF1 unit Port 11 select MSR.		Package
Register Address: E28H, 3624	MSR_R1_PMON_IPERF1_P12	
Uncore R-box 1 PerfMon IPERF1 unit Port 12 select MSR.		Package
Register Address: E29H, 3625	MSR_R1_PMON_IPERF1_P13	
Uncore R-box 1 PerfMon IPERF1 unit Port 13 select MSR.		Package
Register Address: E2AH, 3626	MSR_R1_PMON_IPERF1_P14	
Uncore R-box 1 PerfMon IPERF1 unit Port 14 select MSR.		Package
Register Address: E2BH, 3627	MSR_R1_PMON_IPERF1_P15	
Uncore R-box 1 PerfMon IPERF1 unit Port 15 select MSR.		Package
Register Address: E2CH, 3628	MSR_R1_PMON_QLX_P4	
Uncore R-box 1 PerfMon QLX unit Port 4 select MSR.		Package
Register Address: E2DH, 3629	MSR_R1_PMON_QLX_P5	
Uncore R-box 1 PerfMon QLX unit Port 5 select MSR.		Package
Register Address: E2EH, 3630	MSR_R1_PMON_QLX_P6	
Uncore R-box 1 PerfMon QLX unit Port 6 select MSR.		Package
Register Address: E2FH, 3631	MSR_R1_PMON_QLX_P7	
Uncore R-box 1 PerfMon QLX unit Port 7 select MSR.		Package
Register Address: E30H, 3632	MSR_R1_PMON_EVNT_SEL8	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E31H, 3633	MSR_R1_PMON_CTR8	
Uncore R-box 1 PerfMon counter MSR.		Package

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E32H, 3634	MSR_R1_PMON_EVNT_SEL9	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E33H, 3635	MSR_R1_PMON_CTR9	
Uncore R-box 1 PerfMon counter MSR.		Package
Register Address: E34H, 3636	MSR_R1_PMON_EVNT_SEL10	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E35H, 3637	MSR_R1_PMON_CTR10	
Uncore R-box 1 PerfMon counter MSR.		Package
Register Address: E36H, 3638	MSR_R1_PMON_EVNT_SEL11	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E37H, 3639	MSR_R1_PMON_CTR11	
Uncore R-box 1 PerfMon counter MSR.		Package
Register Address: E38H, 3640	MSR_R1_PMON_EVNT_SEL12	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E39H, 3641	MSR_R1_PMON_CTR12	
Uncore R-box 1 PerfMon counter MSR.		Package
Register Address: E3AH, 3642	MSR_R1_PMON_EVNT_SEL13	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E3BH, 3643	MSR_R1_PMON_CTR13	
Uncore R-box 1PerfMon counter MSR.		Package
Register Address: E3CH, 3644	MSR_R1_PMON_EVNT_SEL14	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E3DH, 3645	MSR_R1_PMON_CTR14	
Uncore R-box 1 PerfMon counter MSR.		Package
Register Address: E3EH, 3646	MSR_R1_PMON_EVNT_SEL15	
Uncore R-box 1 PerfMon event select MSR.		Package
Register Address: E3FH, 3647	MSR_R1_PMON_CTR15	
Uncore R-box 1 PerfMon counter MSR.		Package
Register Address: E45H, 3653	MSR_B0_PMON_MATCH	
Uncore B-box 0 PerfMon local box match MSR.		Package
Register Address: E46H, 3654	MSR_B0_PMON_MASK	
Uncore B-box 0 PerfMon local box mask MSR.		Package
Register Address: E49H, 3657	MSR_S0_PMON_MATCH	
Uncore S-box 0 PerfMon local box match MSR.		Package
Register Address: E4AH, 3658	MSR_S0_PMON_MASK	
Uncore S-box 0 PerfMon local box mask MSR.		Package
Register Address: E4DH, 3661	MSR_B1_PMON_MATCH	

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore B-box 1 PerfMon local box match MSR.		Package
Register Address: E4EH, 3662	MSR_B1_PMON_MASK	
Uncore B-box 1 PerfMon local box mask MSR.		Package
Register Address: E54H, 3668	MSR_M0_PMON_MM_CONFIG	
Uncore M-box 0 PerfMon local box address match/mask config MSR.		Package
Register Address: E55H, 3669	MSR_M0_PMON_ADDR_MATCH	
Uncore M-box 0 PerfMon local box address match MSR.		Package
Register Address: E56H, 3670	MSR_M0_PMON_ADDR_MASK	
Uncore M-box 0 PerfMon local box address mask MSR.		Package
Register Address: E59H, 3673	MSR_S1_PMON_MATCH	
Uncore S-box 1 PerfMon local box match MSR.		Package
Register Address: E5AH, 3674	MSR_S1_PMON_MASK	
Uncore S-box 1 PerfMon local box mask MSR.		Package
Register Address: E5CH, 3676	MSR_M1_PMON_MM_CONFIG	
Uncore M-box 1 PerfMon local box address match/mask config MSR.		Package
Register Address: E5DH, 3677	MSR_M1_PMON_ADDR_MATCH	
Uncore M-box 1 PerfMon local box address match MSR.		Package
Register Address: E5EH, 3678	MSR_M1_PMON_ADDR_MASK	
Uncore M-box 1 PerfMon local box address mask MSR.		Package
Register Address: 3B5H, 965	MSR_UNCORE_PMC5	
See Section 21.3.1.2.2, "Uncore Performance Event Configuration Facility."		Package

2.9 MSRS IN THE INTEL® XEON® PROCESSOR 5600 SERIES BASED ON WESTMERE MICROARCHITECTURE

The Intel® Xeon® Processor 5600 Series is based on Westmere microarchitecture and supports the MSR interfaces listed in Table 2-15, Table 2-16, plus additional MSRs listed in Table 2-18. These MSRs apply to the Intel Core i7, i5, and i3 processor family with a CPUID Signature DisplayFamily_DisplayModel value of 06_25H or 06_2CH; see Table 2-1.

Table 2-18. Additional MSRs Supported by Intel® Processors Based on Westmere Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 13CH, 316	MSR_FEATURE_CONFIG	
AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.		Core

Table 2-18. Additional MSRs Supported by Intel® Processors Based on Westmere Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
1:0	AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.	
63:2	Reserved.	
Register Address: 1A7H, 423	MSR_OFFCORE_RSP_1	
Offcore Response Event Select Register (R/W)		Thread
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
39:32	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 core active.	Package
47:40	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 core active.	Package
63:48	Reserved.	
Register Address: 1B0H, 432	IA32_ENERGY_PERF_BIAS	
See Table 2-2.		Package

2.10 MSRS IN THE INTEL® XEON® PROCESSOR E7 FAMILY BASED ON WESTMERE MICROARCHITECTURE

The Intel® Xeon® Processor E7 Family is based on the Westmere microarchitecture and supports the MSR interfaces listed in Table 2-15 (except MSR address 1ADH), Table 2-16, plus additional MSRs listed in Table 2-19. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2FH.

Table 2-19. Additional MSRs Supported by the Intel® Xeon® Processor E7 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 13CH, 316	MSR_FEATURE_CONFIG	
AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.		Core
1:0	AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.	
63:2	Reserved.	
Register Address: 1A7H, 423	MSR_OFFCORE_RSP_1	
Offcore Response Event Select Register (R/W)		Thread
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Reserved. Attempt to read/write will cause #UD.		Package
Register Address: 1B0H, 432	IA32_ENERGY_PERF_BIAS	
See Table 2-2.		Package
Register Address: F40H, 3904	MSR_C8_PMON_BOX_CTRL	
Uncore C-box 8 PerfMon local box control MSR.		Package
Register Address: F41H, 3905	MSR_C8_PMON_BOX_STATUS	
Uncore C-box 8 PerfMon local box status MSR.		Package
Register Address: F42H, 3906	MSR_C8_PMON_BOX_OVF_CTRL	
Uncore C-box 8 PerfMon local box overflow control MSR.		Package
Register Address: F50H, 3920	MSR_C8_PMON_EVNT_SEL0	
Uncore C-box 8 PerfMon event select MSR.		Package
Register Address: F51H, 3921	MSR_C8_PMON_CTR0	
Uncore C-box 8 PerfMon counter MSR.		Package
Register Address: F52H, 3922	MSR_C8_PMON_EVNT_SEL1	
Uncore C-box 8 PerfMon event select MSR.		Package
Register Address: F53H, 3923	MSR_C8_PMON_CTR1	
Uncore C-box 8 PerfMon counter MSR.		Package
Register Address: F54H, 3924	MSR_C8_PMON_EVNT_SEL2	
Uncore C-box 8 PerfMon event select MSR.		Package
Register Address: F55H, 3925	MSR_C8_PMON_CTR2	
Uncore C-box 8 PerfMon counter MSR.		Package
Register Address: F56H, 3926	MSR_C8_PMON_EVNT_SEL3	
Uncore C-box 8 PerfMon event select MSR.		Package

Table 2-19. Additional MSRs Supported by the Intel® Xeon® Processor E7 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: F57H, 3927	MSR_C8_PMON_CTR3	
Uncore C-box 8 PerfMon counter MSR.		Package
Register Address: F58H, 3928	MSR_C8_PMON_EVNT_SEL4	
Uncore C-box 8 PerfMon event select MSR.		Package
Register Address: F59H, 3929	MSR_C8_PMON_CTR4	
Uncore C-box 8 PerfMon counter MSR.		Package
Register Address: F5AH, 3930	MSR_C8_PMON_EVNT_SEL5	
Uncore C-box 8 PerfMon event select MSR.		Package
Register Address: F5BH, 3931	MSR_C8_PMON_CTR5	
Uncore C-box 8 PerfMon counter MSR.		Package
Register Address: FC0H, 4032	MSR_C9_PMON_BOX_CTRL	
Uncore C-box 9 PerfMon local box control MSR.		Package
Register Address: FC1H, 4033	MSR_C9_PMON_BOX_STATUS	
Uncore C-box 9 PerfMon local box status MSR.		Package
Register Address: FC2H, 4034	MSR_C9_PMON_BOX_OVF_CTRL	
Uncore C-box 9 PerfMon local box overflow control MSR.		Package
Register Address: FD0H, 4048	MSR_C9_PMON_EVNT_SEL0	
Uncore C-box 9 PerfMon event select MSR.		Package
Register Address: FD1H, 4049	MSR_C9_PMON_CTR0	
Uncore C-box 9 PerfMon counter MSR.		Package
Register Address: FD2H, 4050	MSR_C9_PMON_EVNT_SEL1	
Uncore C-box 9 PerfMon event select MSR.		Package
Register Address: FD3H, 4051	MSR_C9_PMON_CTR1	
Uncore C-box 9 PerfMon counter MSR.		Package
Register Address: FD4H, 4052	MSR_C9_PMON_EVNT_SEL2	
Uncore C-box 9 PerfMon event select MSR.		Package
Register Address: FD5H, 4053	MSR_C9_PMON_CTR2	
Uncore C-box 9 PerfMon counter MSR.		Package
Register Address: FD6H, 4054	MSR_C9_PMON_EVNT_SEL3	
Uncore C-box 9 PerfMon event select MSR.		Package
Register Address: FD7H, 4055	MSR_C9_PMON_CTR3	
Uncore C-box 9 PerfMon counter MSR.		Package
Register Address: FD8H, 4056	MSR_C9_PMON_EVNT_SEL4	
Uncore C-box 9 PerfMon event select MSR.		Package
Register Address: FD9H, 4057	MSR_C9_PMON_CTR4	
Uncore C-box 9 PerfMon counter MSR.		Package
Register Address: FDAH, 4058	MSR_C9_PMON_EVNT_SEL5	

Table 2-19. Additional MSRs Supported by the Intel® Xeon® Processor E7 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-box 9 PerfMon event select MSR.		Package
Register Address: FDBH, 4059	MSR_C9_PMON_CTR5	
Uncore C-box 9 PerfMon counter MSR.		Package

2.11 MSRS IN THE INTEL® PROCESSOR FAMILY BASED ON SANDY BRIDGE MICROARCHITECTURE

Table 2-20 lists model-specific registers (MSRs) that are common to the Intel® processor family based on Sandy Bridge microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH or 06_2DH; see Table 2-1. Additional MSRs specific to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH are listed in Table 2-21.

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 0H, 0	IA32_P5_MC_ADDR	
See Section 2.23, "MSRs in Pentium Processors."		Thread
Register Address: 1H, 1	IA32_P5_MC_TYPE	
See Section 2.23, "MSRs in Pentium Processors."		Thread
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.		Thread
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER	
See Section 19.17, "Time-Stamp Counter," and see Table 2-2.		Thread
Register Address: 17H, 23	IA32_PLATFORM_ID	
Platform ID (R) See Table 2-2.		Package
Register Address: 1BH, 27	IA32_APIC_BASE	
See Section 12.4.4, "Local APIC Status and Location," and Table 2-2.		Thread
Register Address: 34H, 52	MSR_SMI_COUNT	
SMI Counter (R/O)		Thread
31:0	SMI Count (R/O) Count SMIs.	
63:32	Reserved.	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64 Processor (R/W) See Table 2-2.		Thread
0	Lock (R/WL)	
1	Enable VMX Inside SMX Operation (R/WL)	
2	Enable VMX Outside SMX Operation (R/WL)	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
14:8	SENTER Local Functions Enables (R/WL)	
15	SENTER Global Functions Enable (R/WL)	
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG	
BIOS Update Trigger Register (W) See Table 2-2.		Core
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Thread
Register Address: C1H, 193	IA32_PMC0	
Performance Counter Register See Table 2-2.		Thread
Register Address: C2H, 194	IA32_PMC1	
Performance Counter Register See Table 2-2.		Thread
Register Address: C3H, 195	IA32_PMC2	
Performance Counter Register See Table 2-2.		Thread
Register Address: C4H, 196	IA32_PMC3	
Performance Counter Register See Table 2-2.		Thread
Register Address: C5H, 197	IA32_PMC4	
Performance Counter Register (if core not shared by threads)		Core
Register Address: C6H, 198	IA32_PMC5	
Performance Counter Register (if core not shared by threads)		Core
Register Address: C7H, 199	IA32_PMC6	
Performance Counter Register (if core not shared by threads)		Core
Register Address: C8H, 200	IA32_PMC7	
Performance Counter Register (if core not shared by threads)		Core
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.	Package
27:16	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.	Package

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
29	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.	Package
39:30	Reserved.	
47:40	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	Package
63:48	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.		Core
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit Note: This field cannot be used to limit package C-state to C3.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.	
24:16	Reserved.	
25	C3 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.	
26	C1 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.	
27	Enable C3 Undemotion (R/W) When set, enables undemotion from demoted C3.	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
28	Enable C1 Undemotion (R/W) When set, enables undemotion from demoted C1.	
63:29	Reserved.	
Register Address: E4H, 228	MSR_PMG_IO_CAPTURE_BASE	
Power Management IO Redirection in C-state (R/W) See http://biosbits.org.		Core
15:0	LVL_2 Base Address (R/W) Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.	
18:16	C-State Range (R/W) Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include. 001b - C6 is the max C-State to include. 010b - C7 is the max C-State to include.	
63:19	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: FEH, 254	IA32_MTRRCAP	
See Table 2-2.		Thread
Register Address: 13CH, 316	MSR_FEATURE_CONFIG	
AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.		Core
1:0	AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.	
63:2	Reserved.	
Register Address: 174H, 372	IA32_SYSENTER_CS	
See Table 2-2.		Thread

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 175H, 373	IA32_SYSENTER_ESP	
See Table 2-2.		Thread
Register Address: 176H, 374	IA32_SYSENTER_EIP	
See Table 2-2.		Thread
Register Address: 179H, 377	IA32_MCG_CAP	
See Table 2-2.		Thread
Register Address: 17AH, 378	IA32_MCG_STATUS	
Global Machine Check Status		Thread
0	RIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.	
1	EIPV When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.	
2	MCIP When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.	
63:3	Reserved.	
Register Address: 186H, 390	IA32_PERFEVTSEL0	
See Table 2-2.		Thread
Register Address: 187H, 391	IA32_PERFEVTSEL1	
See Table 2-2.		Thread
Register Address: 188H, 392	IA32_PERFEVTSEL2	
See Table 2-2.		Thread
Register Address: 189H, 393	IA32_PERFEVTSEL3	
See Table 2-2.		Thread
Register Address: 18AH, 394	IA32_PERFEVTSEL4	
See Table 2-2. If CPUID.0AH:EAX[15:8] > 4.		Core
Register Address: 18BH, 395	IA32_PERFEVTSEL5	
See Table 2-2. If CPUID.0AH:EAX[15:8] > 5.		Core
Register Address: 18CH, 396	IA32_PERFEVTSEL6	
See Table 2-2. If CPUID.0AH:EAX[15:8] > 6.		Core
Register Address: 18DH, 397	IA32_PERFEVTSEL7	
See Table 2-2. If CPUID.0AH:EAX[15:8] > 7.		Core
Register Address: 198H, 408	IA32_PERF_STATUS	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2.		Package
15:0	Current Performance State Value	
63:16	Reserved.	
Register Address: 198H, 408	MSR_PERF_STATUS	
Performance Status		Package
47:32	Core Voltage (R/O) P-state core voltage can be computed by MSR_PERF_STATUS[37:32] * (float) 1/(2^13).	
Register Address: 199H, 409	IA32_PERF_CTL	
See Table 2-2.		Thread
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation (R/W) See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.		Thread
3:0	On demand Clock Modulation Duty Cycle (R/W) In 6.25% increment.	
4	On demand Clock Modulation Enable (R/W)	
63:5	Reserved.	
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) See Table 2-2.		Core
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Core
0	Thermal Status (R/O) See Table 2-2.	
1	Thermal Status Log (R/WC0) See Table 2-2.	
2	PROTCHOT # or FORCEPR# Status (R/O) See Table 2-2.	
3	PROTCHOT # or FORCEPR# Log (R/WC0) See Table 2-2.	
4	Critical Temperature Status (R/O) See Table 2-2.	
5	Critical Temperature Status Log (R/WC0) See Table 2-2.	
6	Thermal Threshold #1 Status (R/O) See Table 2-2.	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
7	Thermal Threshold #1 Log (R/WC0) See Table 2-2.	
8	Thermal Threshold #2 Status (R/O) See Table 2-2.	
9	Thermal Threshold #2 Log (R/WC0) See Table 2-2.	
10	Power Limitation Status (R/O) See Table 2-2.	
11	Power Limitation Log (R/WC0) See Table 2-2.	
15:12	Reserved.	
22:16	Digital Readout (R/O) See Table 2-2.	
26:23	Reserved.	
30:27	Resolution in Degrees Celsius (R/O) See Table 2-2.	
31	Reading Valid (R/O) See Table 2-2.	
63:32	Reserved.	
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		
0	Fast-Strings Enable See Table 2-2.	Thread
6:1	Reserved.	
7	Performance Monitoring Available (R) See Table 2-2.	Thread
10:8	Reserved	
11	Branch Trace Storage Unavailable (R/O) See Table 2-2.	Thread
12	Processor Event Based Sampling Unavailable (R/O) See Table 2-2.	Thread
15:13	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.	Package
18	ENABLE MONITOR FSM (R/W) See Table 2-2.	Thread
21:19	Reserved.	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
22	Limit CPUID Maxval (R/W) See Table 2-2.	Thread
23	xTPR Message Disable (R/W) See Table 2-2.	Thread
33:24	Reserved.	
34	XD Bit Disable (R/W) See Table 2-3.	Thread
37:35	Reserved.	
38	Turbo Mode Disable (R/W) When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.	Package
63:39	Reserved.	
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target		Unique
15:0	Reserved.	
23:16	Temperature Target (R) The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.	
63:24	Reserved.	
Register Address: 1A4H, 420	MSR_MISC_FEATURE_CONTROL	
Miscellaneous Feature Control (R/W)		
0	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.	Core
1	L2 Adjacent Cache Line Prefetcher Disable (R/W) If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).	Core
2	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.	Core
3	DCU IP Prefetcher Disable (R/W) If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.	Core
63:4	Reserved.	
Register Address: 1A6H, 422	MSR_OFFCORE_RSP_0	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Offcore Response Event Select Register (R/W)		Thread
Register Address: 1A7H, 423	MSR_OFFCORE_RSP_1	
Offcore Response Event Select Register (R/W)		Thread
Register Address: 1AAH, 426	MSR_MISC_PWR_MGMT	
Miscellaneous Power Management Control Various model specific features enumeration. See http://biosbits.org.		
Register Address: 1B0H, 432	IA32_ENERGY_PERF_BIAS	
See Table 2-2.		Package
Register Address: 1B1H, 433	IA32_PACKAGE_THERM_STATUS	
See Table 2-2.		Package
Register Address: 1B2H, 434	IA32_PACKAGE_THERM_INTERRUPT	
See Table 2-2.		Package
Register Address: 1C8H, 456	MSR_LBR_SELECT	
Last Branch Record Filtering Select Register (R/W) See Section 19.9.2, "Filtering of Last Branch Records."		Thread
0	CPL_EQ_0	
1	CPL_NEQ_0	
2	JCC	
3	NEAR_REL_CALL	
4	NEAR_IND_CALL	
5	NEAR_RET	
6	NEAR_IND_JMP	
7	NEAR_REL_JMP	
8	FAR_BRANCH	
63:9	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).		Thread
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W) See Table 2-2.		Thread
0	LBR: Last Branch Record	
1	BTF	
5:2	Reserved.	
6	TR: Branch Trace	
7	BTS: Log Branch Trace Message to BTS buffer	
8	BTINT	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
9	BTS_OFF_OS	
10	BTS_OFF_USER	
11	FREEZE_LBR_ON_PMI	
12	FREEZE_PERFMON_ON_PMI	
13	ENABLE_UNCORE_PMI	
14	FREEZE_WHILE_SMM	
63:15	Reserved.	
Register Address: 1DDH, 477	MSR_LER_FROM_LIP	
Last Exception Record From Linear IP (R/W) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Thread
Register Address: 1DEH, 478	MSR_LER_TO_LIP	
Last Exception Record To Linear IP (R/W) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Thread
Register Address: 1F2H, 498	IA32_SMRR_PHYSBASE	
See Table 2-2.		Core
Register Address: 1F3H, 499	IA32_SMRR_PHYSMASK	
See Table 2-2.		Core
Register Address: 1FCH, 508	MSR_POWER_CTL	
See http://biosbits.org.		Core
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0	
See Table 2-2.		Thread
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0	
See Table 2-2.		Thread
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1	
See Table 2-2.		Thread
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1	
See Table 2-2.		Thread
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2	
See Table 2-2.		Thread
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2	
See Table 2-2.		Thread
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3	
See Table 2-2.		Thread
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3	
See Table 2-2.		Thread
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2.		Thread
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4	
See Table 2-2.		Thread
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5	
See Table 2-2.		Thread
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5	
See Table 2-2.		Thread
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6	
See Table 2-2.		Thread
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6	
See Table 2-2.		Thread
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7	
See Table 2-2.		Thread
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7	
See Table 2-2.		Thread
Register Address: 210H, 528	IA32_MTRR_PHYSBASE8	
See Table 2-2.		Thread
Register Address: 211H, 529	IA32_MTRR_PHYSMASK8	
See Table 2-2.		Thread
Register Address: 212H, 530	IA32_MTRR_PHYSBASE9	
See Table 2-2.		Thread
Register Address: 213H, 531	IA32_MTRR_PHYSMASK9	
See Table 2-2.		Thread
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000	
See Table 2-2.		Thread
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000	
See Table 2-2.		Thread
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000	
See Table 2-2.		Thread
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000	
See Table 2-2.		Thread
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000	
See Table 2-2.		Thread
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000	
See Table 2-2.		Thread
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000	
See Table 2-2.		Thread

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000	
See Table 2-2.		Thread
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000	
See Table 2-2.		Thread
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000	
See Table 2-2.		Thread
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000	
See Table 2-2.		Thread
Register Address: 277H, 631	IA32_PAT	
See Table 2-2.		Thread
Register Address: 280H, 640	IA32_MC0_CTL2	
See Table 2-2.		Core
Register Address: 281H, 641	IA32_MC1_CTL2	
See Table 2-2.		Core
Register Address: 282H, 642	IA32_MC2_CTL2	
See Table 2-2.		Core
Register Address: 283H, 643	IA32_MC3_CTL2	
See Table 2-2.		Core
Register Address: 284H, 644	IA32_MC4_CTL2	
Always 0 (CMCI not supported).		Package
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE	
Default Memory Types (R/W) See Table 2-2.		Thread
Register Address: 309H, 777	IA32_FIXED_CTR0	
Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.		Thread
Register Address: 30AH, 778	IA32_FIXED_CTR1	
Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.		Thread
Register Address: 30BH, 779	IA32_FIXED_CTR2	
Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.		Thread
Register Address: 345H, 837	IA32_PERF_CAPABILITIES	
See Table 2-2 and Section 19.4.1, “IA32_DEBUGCTL MSR.”		Thread
5:0	LBR Format See Table 2-2.	
6	PEBS Record Format.	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
7	PEBSSaveArchRegs See Table 2-2.	
11:8	PEBS_REC_FORMAT See Table 2-2.	
12	SMM_FREEZE See Table 2-2.	
63:13	Reserved.	
Register Address: 38DH, 909	IA32_FIXED_CTR_CTRL	
Fixed-Function-Counter Control Register (R/W) See Table 2-2.		Thread
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2 and Section 21.6.2.2, “Global Counter Control Facilities.”		
0	Ovf_PMC0	Thread
1	Ovf_PMC1	Thread
2	Ovf_PMC2	Thread
3	Ovf_PMC3	Thread
4	Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)	Core
5	Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)	Core
6	Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)	Core
7	Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)	Core
31:8	Reserved.	
32	Ovf_FixedCtr0	Thread
33	Ovf_FixedCtr1	Thread
34	Ovf_FixedCtr2	Thread
60:35	Reserved.	
61	Ovf_Uncore	Thread
62	Ovf_BufDSSAVE	Thread
63	CondChgd	Thread
Register Address: 38FH, 911	IA32_PERF_GLOBAL_CTRL	
See Table 2-2 and Section 21.6.2.2, “Global Counter Control Facilities.”		Thread
0	Set 1 to enable PMC0 to count.	Thread
1	Set 1 to enable PMC1 to count.	Thread
2	Set 1 to enable PMC2 to count.	Thread
3	Set 1 to enable PMC3 to count.	Thread
4	Set 1 to enable PMC4 to count (if CPUID.0AH:EAX[15:8] > 4).	Core
5	Set 1 to enable PMC5 to count (if CPUID.0AH:EAX[15:8] > 5).	Core
6	Set 1 to enable PMC6 to count (if CPUID.0AH:EAX[15:8] > 6).	Core
7	Set 1 to enable PMC7 to count (if CPUID.0AH:EAX[15:8] > 7).	Core

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:8	Reserved.	
32	Set 1 to enable FixedCtr0 to count.	Thread
33	Set 1 to enable FixedCtr1 to count.	Thread
34	Set 1 to enable FixedCtr2 to count.	Thread
63:35	Reserved.	
Register Address: 390H, 912	IA32_PERF_GLOBAL_OVF_CTRL	
See Table 2-2 and Section 21.6.2.2, “Global Counter Control Facilities.”		
0	Set 1 to clear Ovf_PMC0.	Thread
1	Set 1 to clear Ovf_PMC1.	Thread
2	Set 1 to clear Ovf_PMC2.	Thread
3	Set 1 to clear Ovf_PMC3.	Thread
4	Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4).	Core
5	Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5).	Core
6	Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6).	Core
7	Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7).	Core
31:8	Reserved.	
32	Set 1 to clear Ovf_FixedCtr0.	Thread
33	Set 1 to clear Ovf_FixedCtr1.	Thread
34	Set 1 to clear Ovf_FixedCtr2.	Thread
60:35	Reserved.	
61	Set 1 to clear Ovf_Uncore.	Thread
62	Set 1 to clear Ovf_BufDSSAVE.	Thread
63	Set 1 to clear CondChgd.	Thread
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Section 21.3.1.1.1, “Processor Event Based Sampling (PEBS).”		Thread
0	Enable PEBS on IA32_PMC0. (R/W)	
1	Enable PEBS on IA32_PMC1. (R/W)	
2	Enable PEBS on IA32_PMC2. (R/W)	
3	Enable PEBS on IA32_PMC3. (R/W)	
31:4	Reserved.	
32	Enable Load Latency on IA32_PMC0. (R/W)	
33	Enable Load Latency on IA32_PMC1. (R/W)	
34	Enable Load Latency on IA32_PMC2. (R/W)	
35	Enable Load Latency on IA32_PMC3. (R/W)	
62:36	Reserved.	
63	Enable Precise Store (R/W)	
Register Address: 3F6H, 1014	MSR_PEBS_LD_LAT	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 21.3.1.1.2, "Load Latency Performance Monitoring Facility."		Thread
15:0	Minimum threshold latency value of tagged load operation that will be counted. (R/W)	
63:36	Reserved.	
Register Address: 3F8H, 1016	MSR_PKG_C3_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C3 Residency Counter (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.	
Register Address: 3F9H, 1017	MSR_PKG_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.	
Register Address: 3FAH, 1018	MSR_PKG_C7_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C7 Residency Counter (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.	
Register Address: 3FCH, 1020	MSR_CORE_C3_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C3 Residency Counter (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.	
Register Address: 3FDH, 1021	MSR_CORE_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.	
Register Address: 3FEH, 1022	MSR_CORE_C7_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C7 Residency Counter (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.	
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Core
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 403H, 1027	IA32_MC0_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Core
Register Address: 406H, 1030	IA32_MC1_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 407H, 1031	IA32_MC1_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 408H, 1032	IA32_MC2_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Core
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 40BH, 1035	IA32_MC2_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 40CH, 1036	IA32_MC3_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 40DH, 1037	IA32_MC3_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Core
Register Address: 40EH, 1038	IA32_MC3_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 40FH, 1039	IA32_MC3_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Core
Register Address: 410H, 1040	IA32_MC4_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
0	PCU Hardware Error (R/W) When set, enables signaling of PCU hardware detected errors.	
1	PCU Controller Error (R/W) When set, enables signaling of PCU controller detected errors.	
2	PCU Firmware Error (R/W) When set, enables signaling of PCU firmware detected errors.	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:2	Reserved.	
Register Address: 411H, 1041	IA32_MC4_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Core
Register Address: 480H, 1152	IA32_VMX_BASIC	
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2 and Appendix A.1, "Basic VMX Information."		Thread
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2 and Appendix A.3, "VM-Execution Controls."		Thread
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Thread
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	
Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2 and Appendix A.4, "VM-Exit Controls."		Thread
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2 and Appendix A.5, "VM-Entry Controls."		Thread
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2 and Appendix A.6, "Miscellaneous Data."		Thread
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2 and Appendix A.7, "VMX-Fixed Bits in CR0."		Thread
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2 and Appendix A.7, "VMX-Fixed Bits in CR0."		Thread
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2 and Appendix A.8, "VMX-Fixed Bits in CR4."		Thread
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2 and Appendix A.8, "VMX-Fixed Bits in CR4."		Thread
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	
Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2 and Appendix A.9, "VMCS Enumeration."		Thread
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2	
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls."		Thread

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 48CH, 1164	IA32_VMX_EPT_VPID_ENUM	
Capability Reporting Register of EPT and VPID (R/O) See Table 2-2		Thread
Register Address: 48DH, 1165	IA32_VMX_TRUE_PINBASED_CTLS	
Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2		Thread
Register Address: 48EH, 1166	IA32_VMX_TRUE_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Table 2-2		Thread
Register Address: 48FH, 1167	IA32_VMX_TRUE_EXIT_CTLS	
Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2		Thread
Register Address: 490H, 1168	IA32_VMX_TRUE_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Flex Controls (R/O) See Table 2-2		Thread
Register Address: 4C1H, 1217	IA32_A_PMC0	
See Table 2-2.		Thread
Register Address: 4C2H, 1218	IA32_A_PMC1	
See Table 2-2.		Thread
Register Address: 4C3H, 1219	IA32_A_PMC2	
See Table 2-2.		Thread
Register Address: 4C4H, 1220	IA32_A_PMC3	
See Table 2-2.		Thread
Register Address: 4C5H, 1221	IA32_A_PMC4	
See Table 2-2.		Core
Register Address: 4C6H, 1222	IA32_A_PMC5	
See Table 2-2.		Core
Register Address: 4C7H, 1223	IA32_A_PMC6	
See Table 2-2.		Core
Register Address: 4C8H, 1224	IA32_A_PMC7	
See Table 2-2.		Core
Register Address: 600H, 1536	IA32_DS_AREA	
DS Save Area (R/W) See Table 2-2 and Section 21.6.3.4, “Debug Store (DS) Mechanism.”		Thread
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers used in RAPL Interfaces (R/O) See Section 16.10.1, “RAPL Interfaces.”		Package
Register Address: 60AH, 1546	MSR_PKGC3_IRTL	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Package C3 Interrupt Response Limit (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
9:0	Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C3 state.	
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	
63:16	Reserved.	
Register Address: 60BH, 1547	MSR_PKGC6_IRTL	
Package C6 Interrupt Response Limit (R/W) This MSR defines the budget allocated for the package to exit from a C6 to a C0 state, where an interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Package
9:0	Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 state.	
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:16	Reserved.	
Register Address: 60DH, 1549	MSR_PKG_C2_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
63:0	Package C2 Residency Counter (R/O) Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.	
Register Address: 610H, 1552	MSR_PKG_POWER_LIMIT	
PKG RAPL Power Limit Control (R/W) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 611H, 1553	MSR_PKG_ENERGY_STATUS	
PKG Energy Status (R/O) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 614H, 1556	MSR_PKG_POWER_INFO	
PKG RAPL Parameters (R/W) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 638H, 1592	MSR_PP0_POWER_LIMIT	
PP0 RAPL Power Limit Control (R/W) See Section 16.10.4, “PP0/PP1 RAPL Domains.”		Package
Register Address: 680H, 1664	MSR_LASTBRANCH_0_FROM_IP	
Last Branch Record 0 From IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.9.1 and record format in Section 19.4.8.1.		Thread
Register Address: 681H, 1665	MSR_LASTBRANCH_1_FROM_IP	
Last Branch Record 1 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 682H, 1666	MSR_LASTBRANCH_2_FROM_IP	
Last Branch Record 2 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 683H, 1667	MSR_LASTBRANCH_3_FROM_IP	
Last Branch Record 3 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 684H, 1668	MSR_LASTBRANCH_4_FROM_IP	
Last Branch Record 4 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 685H, 1669	MSR_LASTBRANCH_5_FROM_IP	
Last Branch Record 5 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 686H, 1670	MSR_LASTBRANCH_6_FROM_IP	
Last Branch Record 6 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 687H, 1671	MSR_LASTBRANCH_7_FROM_IP	
Last Branch Record 7 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 688H, 1672	MSR_LASTBRANCH_8_FROM_IP	
Last Branch Record 8 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 689H, 1673	MSR_LASTBRANCH_9_FROM_IP	
Last Branch Record 9 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68AH, 1674	MSR_LASTBRANCH_10_FROM_IP	
Last Branch Record 10 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68BH, 1675	MSR_LASTBRANCH_11_FROM_IP	
Last Branch Record 11 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68CH, 1676	MSR_LASTBRANCH_12_FROM_IP	
Last Branch Record 12 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68DH, 1677	MSR_LASTBRANCH_13_FROM_IP	
Last Branch Record 13 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68EH, 1678	MSR_LASTBRANCH_14_FROM_IP	
Last Branch Record 14 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 68FH, 1679	MSR_LASTBRANCH_15_FROM_IP	
Last Branch Record 15 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 6C0H, 1728	MSR_LASTBRANCH_0_TO_IP	
Last Branch Record 0 To IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.		Thread
Register Address: 6C1H, 1729	MSR_LASTBRANCH_1_TO_IP	
Last Branch Record 1 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C2H, 1730	MSR_LASTBRANCH_2_TO_IP	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 2 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C3H, 1731	MSR_LASTBRANCH_3_TO_IP	
Last Branch Record 3 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C4H, 1732	MSR_LASTBRANCH_4_TO_IP	
Last Branch Record 4 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C5H, 1733	MSR_LASTBRANCH_5_TO_IP	
Last Branch Record 5 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C6H, 1734	MSR_LASTBRANCH_6_TO_IP	
Last Branch Record 6 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C7H, 1735	MSR_LASTBRANCH_7_TO_IP	
Last Branch Record 7 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C8H, 1736	MSR_LASTBRANCH_8_TO_IP	
Last Branch Record 8 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6C9H, 1737	MSR_LASTBRANCH_9_TO_IP	
Last Branch Record 9 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CAH, 1738	MSR_LASTBRANCH_10_TO_IP	
Last Branch Record 10 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CBH, 1739	MSR_LASTBRANCH_11_TO_IP	
Last Branch Record 11 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CCH, 1740	MSR_LASTBRANCH_12_TO_IP	
Last Branch Record 12 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CDH, 1741	MSR_LASTBRANCH_13_TO_IP	
Last Branch Record 13 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CEH, 1742	MSR_LASTBRANCH_14_TO_IP	
Last Branch Record 14 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6CFH, 1743	MSR_LASTBRANCH_15_TO_IP	

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 15 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6E0H, 1760	IA32_TSC_DEADLINE	
See Table 2-2.		Thread
Register Address: 802H—83FH, 2050—2111	X2APIC MSRs	
See Table 2-2.		Thread
Register Address: C000_0080H	IA32_EFER	
Extended Feature Enables See Table 2-2.		Thread
Register Address: C000_0081H	IA32_STAR	
System Call Target Address (R/W) See Table 2-2.		Thread
Register Address: C000_0082H	IA32_LSTAR	
IA-32e Mode System Call Target Address (R/W) See Table 2-2.		Thread
Register Address: C000_0084H	IA32_FMASK	
System Call Flag Mask (R/W) See Table 2-2.		Thread
Register Address: C000_0100H	IA32_FS_BASE	
Map of BASE Address of FS (R/W) See Table 2-2.		Thread
Register Address: C000_0101H	IA32_GS_BASE	
Map of BASE Address of GS (R/W) See Table 2-2.		Thread
Register Address: C000_0102H	IA32_KERNEL_GS_BASE	
Swap Target of BASE Address of GS (R/W) See Table 2-2.		Thread
Register Address: C000_0103H	IA32_TSC_AUX	
AUXILIARY TSC Signature (R/W) See Table 2-2 and Section 19.17.2, "IA32_TSC_AUX Register and RDTSCP Support."		Thread

2.11.1 MSRs in the 2nd Generation Intel® Core™ Processor Family Based on Sandy Bridge Microarchitecture

Table 2-21 and Table 2-22 list model-specific registers (MSRs) that are specific to the 2nd generation Intel® Core™ processor family based on the Sandy Bridge microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH; see Table 2-1.

Table 2-21. MSRs Supported by the 2nd Generation Intel® Core™ Processors (Sandy Bridge Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
63:32	Reserved.	
Register Address: 60CH, 1548	MSR_PKGC7_IRTL	
Package C7 Interrupt Response Limit (R/W) This MSR defines the budget allocated for the package to exit from a C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Package
9:0	Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C7 state.	
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	
63:16	Reserved.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 63AH, 1594	MSR_PP0_POLICY	

Table 2-21. MSRs Supported by the 2nd Generation Intel® Core™ Processors (Sandy Bridge Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
PP0 Balance Policy (R/W) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 640H, 1600	MSR_PP1_POWER_LIMIT	
PP1 RAPL Power Limit Control (R/W) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 641H, 1601	MSR_PP1_ENERGY_STATUS	
PP1 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 642H, 1602	MSR_PP1_POLICY	
PP1 Balance Policy (R/W) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
See Table 2-20, Table 2-21, and Table 2-22 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH.		

Table 2-22 lists the MSRs of uncore PMU for Intel processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH.

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 391H, 913	MSR_UNC_PERF_GLOBAL_CTRL	
Uncore PMU Global Control		Package
0	Slice 0 select.	
1	Slice 1 select.	
2	Slice 2 select.	
3	Slice 3 select.	
4	Slice 4 select.	
18:5	Reserved.	
29	Enable all uncore counters.	
30	Enable wake on PMI.	
31	Enable Freezing counter when overflow.	
63:32	Reserved.	
Register Address: 392H, 914	MSR_UNC_PERF_GLOBAL_STATUS	
Uncore PMU Main Status		Package
0	Fixed counter overflowed.	
1	An ARB counter overflowed.	
2	Reserved.	
3	A CBox counter overflowed (on any slice).	
63:4	Reserved.	

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 394H, 916	MSR_UNC_PERF_FIXED_CTRL	
Uncore Fixed Counter Control (R/W)		Package
19:0	Reserved.	
20	Enable overflow propagation.	
21	Reserved.	
22	Enable counting.	
63:23	Reserved.	
Register Address: 395H, 917	MSR_UNC_PERF_FIXED_CTR	
Uncore Fixed Counter		Package
47:0	Current count.	
63:48	Reserved.	
Register Address: 396H, 918	MSR_UNC_CBO_CONFIG	
Uncore C-Box Configuration Information (R/O)		Package
3:0	Report the number of C-Box units with performance counters, including processor cores and processor graphics.	
63:4	Reserved.	
Register Address: 3B0H, 946	MSR_UNC_ARB_PERFCTR0	
Uncore Arb Unit, Performance Counter 0		Package
Register Address: 3B1H, 947	MSR_UNC_ARB_PERFCTR1	
Uncore Arb Unit, Performance Counter 1		Package
Register Address: 3B2H, 944	MSR_UNC_ARB_PERFEVTSEL0	
Uncore Arb Unit, Counter 0 Event Select MSR		Package
Register Address: 3B3H, 945	MSR_UNC_ARB_PERFEVTSEL1	
Uncore Arb unit, Counter 1 Event Select MSR		Package
Register Address: 700H, 1792	MSR_UNC_CBO_0_PERFEVTSEL0	
Uncore C-Box 0, Counter 0 Event Select MSR		Package
Register Address: 701H, 1793	MSR_UNC_CBO_0_PERFEVTSEL1	
Uncore C-Box 0, Counter 1 Event Select MSR		Package
Register Address: 702H, 1794	MSR_UNC_CBO_0_PERFEVTSEL2	
Uncore C-Box 0, Counter 2 Event Select MSR		Package
Register Address: 703H, 1795	MSR_UNC_CBO_0_PERFEVTSEL3	
Uncore C-Box 0, Counter 3 Event Select MSR		Package
Register Address: 705H, 1797	MSR_UNC_CBO_0_UNIT_STATUS	
Uncore C-Box 0, Unit Status for Counter 0-3		Package
Register Address: 706H, 1798	MSR_UNC_CBO_0_PERFCTR0	
Uncore C-Box 0, Performance Counter 0		Package
Register Address: 707H, 1799	MSR_UNC_CBO_0_PERFCTR1	
Uncore C-Box 0, Performance Counter 1		Package

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 708H, 1800	MSR_UNC_CBO_0_PERFCTR2	
Uncore C-Box 0, Performance Counter 2		Package
Register Address: 709H, 1801	MSR_UNC_CBO_0_PERFCTR3	
Uncore C-Box 0, Performance Counter 3		Package
Register Address: 710H, 1808	MSR_UNC_CBO_1_PERFEVTSEL0	
Uncore C-Box 1, Counter 0 Event Select MSR		Package
Register Address: 711H, 1809	MSR_UNC_CBO_1_PERFEVTSEL1	
Uncore C-Box 1, Counter 1 Event Select MSR		Package
Register Address: 712H, 1810	MSR_UNC_CBO_1_PERFEVTSEL2	
Uncore C-Box 1, Counter 2 Event Select MSR		Package
Register Address: 713H, 1811	MSR_UNC_CBO_1_PERFEVTSEL3	
Uncore C-Box 1, Counter 3 Event Select MSR		Package
Register Address: 715H, 1813	MSR_UNC_CBO_1_UNIT_STATUS	
Uncore C-Box 1, Unit Status for Counter 0-3		Package
Register Address: 716H, 1814	MSR_UNC_CBO_1_PERFCTR0	
Uncore C-Box 1, Performance Counter 0		Package
Register Address: 717H, 1815	MSR_UNC_CBO_1_PERFCTR1	
Uncore C-Box 1, Performance Counter 1		Package
Register Address: 718H, 1816	MSR_UNC_CBO_1_PERFCTR2	
Uncore C-Box 1, Performance Counter 2		Package
Register Address: 719H, 1817	MSR_UNC_CBO_1_PERFCTR3	
Uncore C-Box 1, Performance Counter 3		Package
Register Address: 720H, 1824	MSR_UNC_CBO_2_PERFEVTSEL0	
Uncore C-Box 2, Counter 0 Event Select MSR		Package
Register Address: 721H, 1825	MSR_UNC_CBO_2_PERFEVTSEL1	
Uncore C-Box 2, Counter 1 Event Select MSR		Package
Register Address: 722H, 1826	MSR_UNC_CBO_2_PERFEVTSEL2	
Uncore C-Box 2, Counter 2 Event Select MSR		Package
Register Address: 723H, 1827	MSR_UNC_CBO_2_PERFEVTSEL3	
Uncore C-Box 2, Counter 3 Event Select MSR		Package
Register Address: 725H, 1829	MSR_UNC_CBO_2_UNIT_STATUS	
Uncore C-Box 2, Unit Status for Counter 0-3		Package
Register Address: 726H, 1830	MSR_UNC_CBO_2_PERFCTR0	
Uncore C-Box 2, Performance Counter 0		Package
Register Address: 727H, 1831	MSR_UNC_CBO_2_PERFCTR1	
Uncore C-Box 2, Performance Counter 1		Package
Register Address: 728H, 1832	MSR_UNC_CBO_3_PERFCTR2	

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 3, Performance Counter 2		Package
Register Address: 729H, 1833	MSR_UNC_CBO_3_PERFCTR3	
Uncore C-Box 3, Performance Counter 3		Package
Register Address: 730H, 1840	MSR_UNC_CBO_3_PERFEVTSEL0	
Uncore C-Box 3, Counter 0 Event Select MSR		Package
Register Address: 731H, 1841	MSR_UNC_CBO_3_PERFEVTSEL1	
Uncore C-Box 3, Counter 1 Event Select MSR		Package
Register Address: 732H, 1842	MSR_UNC_CBO_3_PERFEVTSEL2	
Uncore C-Box 3, Counter 2 Event Select MSR		Package
Register Address: 733H, 1843	MSR_UNC_CBO_3_PERFEVTSEL3	
Uncore C-Box 3, counter 3 Event Select MSR		Package
Register Address: 735H, 1845	MSR_UNC_CBO_3_UNIT_STATUS	
Uncore C-Box 3, Unit Status for Counter 0-3		Package
Register Address: 736H, 1846	MSR_UNC_CBO_3_PERFCTR0	
Uncore C-Box 3, Performance Counter 0		Package
Register Address: 737H, 1847	MSR_UNC_CBO_3_PERFCTR1	
Uncore C-Box 3, Performance Counter 1		Package
Register Address: 738H, 1848	MSR_UNC_CBO_3_PERFCTR2	
Uncore C-Box 3, Performance Counter 2		Package
Register Address: 739H, 1849	MSR_UNC_CBO_3_PERFCTR3	
Uncore C-Box 3, Performance Counter 3		Package
Register Address: 740H, 1856	MSR_UNC_CBO_4_PERFEVTSEL0	
Uncore C-Box 4, Counter 0 Event Select MSR		Package
Register Address: 741H, 1857	MSR_UNC_CBO_4_PERFEVTSEL1	
Uncore C-Box 4, Counter 1 Event Select MSR		Package
Register Address: 742H, 1858	MSR_UNC_CBO_4_PERFEVTSEL2	
Uncore C-Box 4, Counter 2 Event Select MSR		Package
Register Address: 743H, 1859	MSR_UNC_CBO_4_PERFEVTSEL3	
Uncore C-Box 4, Counter 3 Event Select MSR		Package
Register Address: 745H, 1861	MSR_UNC_CBO_4_UNIT_STATUS	
Uncore C-Box 4, Unit status for Counter 0-3		Package
Register Address: 746H, 1862	MSR_UNC_CBO_4_PERFCTR0	
Uncore C-Box 4, Performance Counter 0		Package
Register Address: 747H, 1863	MSR_UNC_CBO_4_PERFCTR1	
Uncore C-Box 4, Performance Counter 1		Package
Register Address: 748H, 1864	MSR_UNC_CBO_4_PERFCTR2	
Uncore C-Box 4, Performance Counter 2		Package

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 749H, 1865	MSR_UNC_CBO_4_PERFCTR3	
Uncore C-Box 4, Performance Counter 3		Package

2.11.2 MSRs in the Intel® Xeon® Processor E5 Family Based on Sandy Bridge Microarchitecture

Table 2-23 lists additional model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 Family based on Sandy Bridge microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2DH, and also support MSRs listed in Table 2-20 and Table 2-24.

Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 17FH, 383	MSR_ERROR_CONTROL	
	MC Bank Error Configuration (R/W)	Package
0	Reserved.	
1	MemError Log Enable (R/W) When set, enables IMC status bank to log additional info in bits 36:32.	
63:2	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 cores active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 cores active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 cores active.	Package
39:32	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 cores active.	Package
47:40	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 cores active.	Package
55:48	Maximum Ratio Limit for 7C Maximum turbo ratio limit of 7 cores active.	Package
63:56	Maximum Ratio Limit for 8C Maximum turbo ratio limit of 8 cores active.	Package
Register Address: 285H, 645	IA32_MC5_CTL2	
See Table 2-2.		Package

Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 286H, 646	IA32_MC6_CTL2	
See Table 2-2.		Package
Register Address: 287H, 647	IA32_MC7_CTL2	
See Table 2-2.		Package
Register Address: 288H, 648	IA32_MC8_CTL2	
See Table 2-2.		Package
Register Address: 289H, 649	IA32_MC9_CTL2	
See Table 2-2.		Package
Register Address: 28AH, 650	IA32_MC10_CTL2	
See Table 2-2.		Package
Register Address: 28BH, 651	IA32_MC11_CTL2	
See Table 2-2.		Package
Register Address: 28CH, 652	IA32_MC12_CTL2	
See Table 2-2.		Package
Register Address: 28DH, 653	IA32_MC13_CTL2	
See Table 2-2.		Package
Register Address: 28EH, 654	IA32_MC14_CTL2	
See Table 2-2.		Package
Register Address: 28FH, 655	IA32_MC15_CTL2	
See Table 2-2.		Package
Register Address: 290H, 656	IA32_MC16_CTL2	
See Table 2-2.		Package
Register Address: 291H, 657	IA32_MC17_CTL2	
See Table 2-2.		Package
Register Address: 292H, 658	IA32_MC18_CTL2	
See Table 2-2.		Package
Register Address: 293H, 659	IA32_MC19_CTL2	
See Table 2-2.		Package
Register Address: 39CH, 924	MSR_PEBS_NUM_ALT	
ENABLE_PEBS_NUM_ALT (R/W)		Package
0	ENABLE_PEBS_NUM_ALT (R/W) Write 1 to enable alternate PEBS counting logic for specific events requiring additional configuration, see https://perfmon-events.intel.com/.	
63:1	Reserved, must be zero.	
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Package
Register Address: 415H, 1045	IA32_MC5_STATUS	

Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 417H, 1047	IA32_MC5_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 41BH, 1051	IA32_MC6_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 41CH, 1052	IA32_MC7_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 41DH, 1053	IA32_MC7_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 41EH, 1054	IA32_MC7_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 41FH, 1055	IA32_MC7_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 420H, 1056	IA32_MC8_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 421H, 1057	IA32_MC8_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 422H, 1058	IA32_MC8_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 423H, 1059	IA32_MC8_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 424H, 1060	IA32_MC9_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 425H, 1061	IA32_MC9_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 426H, 1062	IA32_MC9_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 427H, 1063	IA32_MC9_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package

Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 428H, 1064	IA32_MC10_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Package
Register Address: 429H, 1065	IA32_MC10_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 18.		Package
Register Address: 42AH, 1066	IA32_MC10_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.”		Package
Register Address: 42BH, 1067	IA32_MC10_MISC	
See Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.”		Package
Register Address: 42CH, 1068	IA32_MC11_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Package
Register Address: 42DH, 1069	IA32_MC11_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 18.		Package
Register Address: 42EH, 1070	IA32_MC11_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.”		Package
Register Address: 42FH, 1071	IA32_MC11_MISC	
See Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.”		Package
Register Address: 430H, 1072	IA32_MC12_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Package
Register Address: 431H, 1073	IA32_MC12_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 18.		Package
Register Address: 432H, 1074	IA32_MC12_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.”		Package
Register Address: 433H, 1075	IA32_MC12_MISC	
See Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.”		Package
Register Address: 434H, 1076	IA32_MC13_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Package
Register Address: 435H, 1077	IA32_MC13_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 18.		Package
Register Address: 436H, 1078	IA32_MC13_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.”		Package
Register Address: 437H, 1079	IA32_MC13_MISC	
See Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.”		Package
Register Address: 438H, 1080	IA32_MC14_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Package
Register Address: 439H, 1081	IA32_MC14_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 18.		Package
Register Address: 43AH, 1082	IA32_MC14_ADDR	

Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 43BH, 1083	IA32_MC14_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 43CH, 1084	IA32_MC15_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 43DH, 1085	IA32_MC15_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 43EH, 1086	IA32_MC15_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 43FH, 1087	IA32_MC15_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 440H, 1088	IA32_MC16_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 441H, 1089	IA32_MC16_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 442H, 1090	IA32_MC16_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 443H, 1091	IA32_MC16_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 444H, 1092	IA32_MC17_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 445H, 1093	IA32_MC17_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 446H, 1094	IA32_MC17_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 447H, 1095	IA32_MC17_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 448H, 1096	IA32_MC18_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 449H, 1097	IA32_MC18_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 44AH, 1098	IA32_MC18_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 44BH, 1099	IA32_MC18_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 44CH, 1100	IA32_MC19_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package

Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 44DH, 1101	IA32_MC19_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 18.		Package
Register Address: 44EH, 1102	IA32_MC19_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 44FH, 1103	IA32_MC19_MISC	
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs."		Package
Register Address: 613H, 1555	MSR_PKG_PERF_STATUS	
Package RAPL Perf Status (R/O)		Package
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
DRAM RAPL Power Limit Control (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	
DRAM RAPL Parameters (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
See Table 2-20, Table 2-23, and Table 2-24 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_2DH.		

2.11.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 Family

Intel Xeon Processor E5 family is based on the Sandy Bridge microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-24. For complete details of the uncore PMU, refer to the Intel Xeon Processor E5 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2DH.

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: C08H, 3080	MSR_U_PMON_UCLK_FIXED_CTL	
Uncore U-box UCLK Fixed Counter Control		Package
Register Address: C09H, 3081	MSR_U_PMON_UCLK_FIXED_CTR	

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore U-box UCLK Fixed Counter		Package
Register Address: C10H, 3088	MSR_U_PMON_EVNTSEL0	
Uncore U-box PerfMon Event Select for U-box Counter 0		Package
Register Address: C11H, 3089	MSR_U_PMON_EVNTSEL1	
Uncore U-box PerfMon Event Select for U-box Counter 1		Package
Register Address: C16H, 3094	MSR_U_PMON_CTR0	
Uncore U-box PerfMon Counter 0		Package
Register Address: C17H, 3095	MSR_U_PMON_CTR1	
Uncore U-box PerfMon Counter 1		Package
Register Address: C24H, 3108	MSR_PCU_PMON_BOX_CTL	
Uncore PCU PerfMon for PCU-box-wide Control		Package
Register Address: C30H, 3120	MSR_PCU_PMON_EVNTSEL0	
Uncore PCU PerfMon Event Select for PCU Counter 0		Package
Register Address: C31H, 3121	MSR_PCU_PMON_EVNTSEL1	
Uncore PCU PerfMon Event Select for PCU Counter 1		Package
Register Address: C32H, 3122	MSR_PCU_PMON_EVNTSEL2	
Uncore PCU PerfMon Event Select for PCU Counter 2		Package
Register Address: C33H, 3123	MSR_PCU_PMON_EVNTSEL3	
Uncore PCU PerfMon Event Select for PCU Counter 3		Package
Register Address: C34H, 3124	MSR_PCU_PMON_BOX_FILTER	
Uncore PCU PerfMon box-wide Filter		Package
Register Address: C36H, 3126	MSR_PCU_PMON_CTR0	
Uncore PCU PerfMon Counter 0		Package
Register Address: C37H, 3127	MSR_PCU_PMON_CTR1	
Uncore PCU PerfMon Counter 1		Package
Register Address: C38H, 3128	MSR_PCU_PMON_CTR2	
Uncore PCU PerfMon Counter 2		Package
Register Address: C39H, 3129	MSR_PCU_PMON_CTR3	
Uncore PCU PerfMon Counter 3		Package
Register Address: D04H, 3332	MSR_C0_PMON_BOX_CTL	
Uncore C-box 0 PerfMon Local Box Wide Control		Package
Register Address: D10H, 3344	MSR_C0_PMON_EVNTSEL0	
Uncore C-box 0 PerfMon Event Select for C-box 0 Counter 0		Package
Register Address: D11H, 3345	MSR_C0_PMON_EVNTSEL1	
Uncore C-box 0 PerfMon Event Select for C-box 0 Counter 1		Package
Register Address: D12H, 3346	MSR_C0_PMON_EVNTSEL2	
Uncore C-box 0 PerfMon Event Select for C-box 0 Counter 2		Package

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: D13H, 3347	MSR_C0_PMON_EVNTSEL3	
Uncore C-box 0 PerfMon Event Select for C-box 0 Counter 3		Package
Register Address: D14H, 3348	MSR_C0_PMON_BOX_FILTER	
Uncore C-box 0 PerfMon Box Wide Filter		Package
Register Address: D16H, 3350	MSR_C0_PMON_CTR0	
Uncore C-box 0 PerfMon Counter 0		Package
Register Address: D17H, 3351	MSR_C0_PMON_CTR1	
Uncore C-box 0 PerfMon Counter 1		Package
Register Address: D18H, 3352	MSR_C0_PMON_CTR2	
Uncore C-box 0 PerfMon Counter 2		Package
Register Address: D19H, 3353	MSR_C0_PMON_CTR3	
Uncore C-box 0 PerfMon Counter 3		Package
Register Address: D24H, 3364	MSR_C1_PMON_BOX_CTL	
Uncore C-box 1 PerfMon Local Box Wide Control		Package
Register Address: D30H, 3376	MSR_C1_PMON_EVNTSEL0	
Uncore C-box 1 PerfMon Event Select for C-box 1 Counter 0		Package
Register Address: D31H, 3377	MSR_C1_PMON_EVNTSEL1	
Uncore C-box 1 PerfMon Event Select for C-box 1 Counter 1		Package
Register Address: D32H, 3378	MSR_C1_PMON_EVNTSEL2	
Uncore C-box 1 PerfMon Event Select for C-box 1 Counter 2		Package
Register Address: D33H, 3379	MSR_C1_PMON_EVNTSEL3	
Uncore C-box 1 PerfMon Event Select for C-box 1 Counter 3		Package
Register Address: D34H, 3380	MSR_C1_PMON_BOX_FILTER	
Uncore C-box 1 PerfMon Box Wide Filter		Package
Register Address: D36H, 3382	MSR_C1_PMON_CTR0	
Uncore C-box 1 PerfMon Counter 0		Package
Register Address: D37H, 3383	MSR_C1_PMON_CTR1	
Uncore C-box 1 PerfMon Counter 1		Package
Register Address: D38H, 3384	MSR_C1_PMON_CTR2	
Uncore C-box 1 PerfMon Counter 2		Package
Register Address: D39H, 3385	MSR_C1_PMON_CTR3	
Uncore C-box 1 PerfMon Counter 3		Package
Register Address: D44H, 3396	MSR_C2_PMON_BOX_CTL	
Uncore C-box 2 PerfMon Local Box Wide Control		Package
Register Address: D50H, 3408	MSR_C2_PMON_EVNTSEL0	
Uncore C-box 2 PerfMon Event Select for C-box 2 Counter 0		Package
Register Address: D51H, 3409	MSR_C2_PMON_EVNTSEL1	

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-box 2 PerfMon Event Select for C-box 2 Counter 1		Package
Register Address: D52H, 3410	MSR_C2_PMON_EVNTSEL2	
Uncore C-box 2 PerfMon Event Select for C-box 2 Counter 2		Package
Register Address: D53H, 3411	MSR_C2_PMON_EVNTSEL3	
Uncore C-box 2 PerfMon Event Select for C-box 2 Counter 3		Package
Register Address: D54H, 3412	MSR_C2_PMON_BOX_FILTER	
Uncore C-box 2 PerfMon Box Wide Filter		Package
Register Address: D56H, 3414	MSR_C2_PMON_CTR0	
Uncore C-box 2 PerfMon Counter 0		Package
Register Address: D57H, 3415	MSR_C2_PMON_CTR1	
Uncore C-box 2 PerfMon Counter 1		Package
Register Address: D58H, 3416	MSR_C2_PMON_CTR2	
Uncore C-box 2 PerfMon Counter 2		Package
Register Address: D59H, 3417	MSR_C2_PMON_CTR3	
Uncore C-box 2 PerfMon Counter 3		Package
Register Address: D64H, 3428	MSR_C3_PMON_BOX_CTL	
Uncore C-box 3 PerfMon Local Box Wide Control		Package
Register Address: D70H, 3440	MSR_C3_PMON_EVNTSEL0	
Uncore C-box 3 PerfMon Event Select for C-box 3 Counter 0		Package
Register Address: D71H, 3441	MSR_C3_PMON_EVNTSEL1	
Uncore C-box 3 PerfMon Event Select for C-box 3 Counter 1		Package
Register Address: D72H, 3442	MSR_C3_PMON_EVNTSEL2	
Uncore C-box 3 PerfMon Event Select for C-box 3 Counter 2		Package
Register Address: D73H, 3443	MSR_C3_PMON_EVNTSEL3	
Uncore C-box 3 PerfMon Event Select for C-box 3 Counter 3		Package
Register Address: D74H, 3444	MSR_C3_PMON_BOX_FILTER	
Uncore C-box 3 PerfMon Box Wide Filter		Package
Register Address: D76H, 3446	MSR_C3_PMON_CTR0	
Uncore C-box 3 PerfMon Counter 0		Package
Register Address: D77H, 3447	MSR_C3_PMON_CTR1	
Uncore C-box 3 PerfMon Counter 1		Package
Register Address: D78H, 3448	MSR_C3_PMON_CTR2	
Uncore C-box 3 PerfMon Counter 2		Package
Register Address: D79H, 3449	MSR_C3_PMON_CTR3	
Uncore C-box 3 PerfMon Counter 3		Package
Register Address: D84H, 3460	MSR_C4_PMON_BOX_CTL	
Uncore C-box 4 PerfMon Local Box Wide Control		Package

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: D90H, 3472	MSR_C4_PMON_EVNTSEL0	
Uncore C-box 4 PerfMon Event Select for C-box 4 Counter 0		Package
Register Address: D91H, 3473	MSR_C4_PMON_EVNTSEL1	
Uncore C-box 4 PerfMon Event Select for C-box 4 Counter 1		Package
Register Address: D92H, 3474	MSR_C4_PMON_EVNTSEL2	
Uncore C-box 4 PerfMon Event Select for C-box 4 Counter 2		Package
Register Address: D93H, 3475	MSR_C4_PMON_EVNTSEL3	
Uncore C-box 4 PerfMon Event Select for C-box 4 Counter 3		Package
Register Address: D94H, 3476	MSR_C4_PMON_BOX_FILTER	
Uncore C-box 4 PerfMon Box Wide Filter		Package
Register Address: D96H, 3478	MSR_C4_PMON_CTR0	
Uncore C-box 4 PerfMon Counter 0		Package
Register Address: D97H, 3479	MSR_C4_PMON_CTR1	
Uncore C-box 4 PerfMon Counter 1		Package
Register Address: D98H, 3480	MSR_C4_PMON_CTR2	
Uncore C-box 4 PerfMon Counter 2		Package
Register Address: D99H, 3481	MSR_C4_PMON_CTR3	
Uncore C-box 4 PerfMon Counter 3		Package
Register Address: DA4H, 3492	MSR_C5_PMON_BOX_CTL	
Uncore C-box 5 PerfMon Local Box Wide Control		Package
Register Address: DB0H, 3504	MSR_C5_PMON_EVNTSEL0	
Uncore C-box 5 PerfMon Event Select for C-box 5 Counter 0		Package
Register Address: DB1H, 3505	MSR_C5_PMON_EVNTSEL1	
Uncore C-box 5 PerfMon Event Select for C-box 5 Counter 1		Package
Register Address: DB2H, 3506	MSR_C5_PMON_EVNTSEL2	
Uncore C-box 5 PerfMon Event Select for C-box 5 Counter 2		Package
Register Address: DB3H, 3507	MSR_C5_PMON_EVNTSEL3	
Uncore C-box 5 PerfMon Event Select for C-box 5 Counter 3		Package
Register Address: DB4H, 3508	MSR_C5_PMON_BOX_FILTER	
Uncore C-box 5 PerfMon Box Wide Filter		Package
Register Address: DB6H, 3510	MSR_C5_PMON_CTR0	
Uncore C-box 5 PerfMon Counter 0		Package
Register Address: DB7H, 3511	MSR_C5_PMON_CTR1	
Uncore C-box 5 PerfMon Counter 1		Package
Register Address: DB8H, 3512	MSR_C5_PMON_CTR2	
Uncore C-box 5 PerfMon Counter 2		Package
Register Address: DB9H, 3513	MSR_C5_PMON_CTR3	

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-box 5 PerfMon Counter 3		Package
Register Address: DC4H, 3524	MSR_C6_PMON_BOX_CTL	
Uncore C-box 6 PerfMon Local Box Wide Control		Package
Register Address: DD0H, 3536	MSR_C6_PMON_EVNTSEL0	
Uncore C-box 6 PerfMon Event Select for C-box 6 Counter 0		Package
Register Address: DD1H, 3537	MSR_C6_PMON_EVNTSEL1	
Uncore C-box 6 PerfMon Event Select for C-box 6 Counter 1		Package
Register Address: DD2H, 3538	MSR_C6_PMON_EVNTSEL2	
Uncore C-box 6 PerfMon Event Select for C-box 6 Counter 2		Package
Register Address: DD3H, 3539	MSR_C6_PMON_EVNTSEL3	
Uncore C-box 6 PerfMon Event Select for C-box 6 Counter 3		Package
Register Address: DD4H, 3540	MSR_C6_PMON_BOX_FILTER	
Uncore C-box 6 PerfMon Box Wide Filter		Package
Register Address: DD6H, 3542	MSR_C6_PMON_CTR0	
Uncore C-box 6 PerfMon Counter 0		Package
Register Address: DD7H, 3543	MSR_C6_PMON_CTR1	
Uncore C-box 6 PerfMon Counter 1		Package
Register Address: DD8H, 3544	MSR_C6_PMON_CTR2	
Uncore C-box 6 PerfMon Counter 2		Package
Register Address: DD9H, 3545	MSR_C6_PMON_CTR3	
Uncore C-box 6 PerfMon Counter 3		Package
Register Address: DE4H, 3556	MSR_C7_PMON_BOX_CTL	
Uncore C-box 7 PerfMon Local Box Wide Control		Package
Register Address: DF0H, 3568	MSR_C7_PMON_EVNTSEL0	
Uncore C-box 7 PerfMon Event Select for C-box 7 Counter 0		Package
Register Address: DF1H, 3569	MSR_C7_PMON_EVNTSEL1	
Uncore C-box 7 PerfMon Event Select for C-box 7 Counter 1		Package
Register Address: DF2H, 3570	MSR_C7_PMON_EVNTSEL2	
Uncore C-box 7 PerfMon Event Select for C-box 7 Counter 2		Package
Register Address: DF3H, 3571	MSR_C7_PMON_EVNTSEL3	
Uncore C-box 7 PerfMon Event Select for C-box 7 Counter 3		Package
Register Address: DF4H, 3572	MSR_C7_PMON_BOX_FILTER	
Uncore C-box 7 PerfMon Box Wide Filter		Package
Register Address: DF6H, 3574	MSR_C7_PMON_CTR0	
Uncore C-box 7 PerfMon Counter 0		Package
Register Address: DF7H, 3575	MSR_C7_PMON_CTR1	
Uncore C-box 7 PerfMon Counter 1		Package

Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: DF8H, 3576	MSR_C7_PMON_CTR2	
Uncore C-box 7 PerfMon Counter 2		Package
Register Address: DF9H, 3577	MSR_C7_PMON_CTR3	
Uncore C-box 7 PerfMon Counter 3		Package

2.12 MSRS IN THE 3RD GENERATION INTEL® CORE™ PROCESSOR FAMILY BASED ON IVY BRIDGE MICROARCHITECTURE

The 3rd generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v2 product family based on Ivy Bridge microarchitecture support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, and Table 2-25. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3AH.

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors Based on Ivy Bridge Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.	Package
27:16	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.	Package
29	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates that TDP Limit for Turbo mode is not programmable.	Package
31:30	Reserved.	
32	Low Power Mode Support (LPM) (R/O) When set to 1, indicates that LPM is supported. When set to 0, indicates LPM is not supported.	Package
34:33	Number of ConfigTDP Levels (R/O) 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 03: Reserved	Package
39:35	Reserved.	

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors Based on Ivy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
47:40	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	Package
55:48	Minimum Operating Ratio (R/O) Contains the minimum supported operating ratio in units of 100 MHz.	Package
63:56	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
	C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.	Core
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.	
24:16	Reserved	
25	C3 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.	
26	C1 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.	
27	Enable C3 Undemotion (R/W) When set, enables undemotion from demoted C3.	

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors Based on Ivy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
28	Enable C1 Undemotion (R/W) When set, enables undemotion from demoted C1.	
63:29	Reserved.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 648H, 1608	MSR_CONFIG_TDP_NOMINAL	
Base TDP Ratio (R/O)		Package
7:0	Config_TDP_Base Base TDP level ratio to be used for this specific processor (in units of 100 MHz).	
63:8	Reserved.	
Register Address: 649H, 1609	MSR_CONFIG_TDP_LEVEL1	
	ConfigTDP Level 1 ratio and power level (R/O)	Package
14:0	PKG_TDP_LVL1 Power setting for ConfigTDP Level 1.	
15	Reserved.	
23:16	Config_TDP_LVL1_Ratio ConfigTDP level 1 ratio to be used for this specific processor.	
31:24	Reserved.	
46:32	PKG_MAX_PWR_LVL1 Max Power setting allowed for ConfigTDP Level 1.	
47	Reserved.	
62:48	PKG_MIN_PWR_LVL1 MIN Power setting allowed for ConfigTDP Level 1.	
63	Reserved.	
Register Address: 64AH, 1610	MSR_CONFIG_TDP_LEVEL2	
ConfigTDP Level 2 ratio and power level (R/O)		Package
14:0	PKG_TDP_LVL2 Power setting for ConfigTDP Level 2.	
15	Reserved.	
23:16	Config_TDP_LVL2_Ratio ConfigTDP level 2 ratio to be used for this specific processor.	
31:24	Reserved.	
46:32	PKG_MAX_PWR_LVL2 Max Power setting allowed for ConfigTDP Level 2.	
47	Reserved.	

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors Based on Ivy Bridge Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
62:48	PKG_MIN_PWR_LVL2 MIN Power setting allowed for ConfigTDP Level 2.	
63	Reserved.	
Register Address: 64BH, 1611	MSR_CONFIG_TDP_CONTROL	
ConfigTDP Control (R/W)		Package
1:0	TDP_LEVEL (RW/L) System BIOS can program this field.	
30:2	Reserved.	
31	Config_TDP_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.	
63:32	Reserved.	
Register Address: 64CH, 1612	MSR_TURBO_ACTIVATION_RATIO	
ConfigTDP Control (R/W)		Package
7:0	MAX_NON_TURBO_RATIO (RW/L) System BIOS can program this field.	
30:8	Reserved.	
31	TURBO_ACTIVATION_RATIO_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.	
63:32	Reserved.	
See Table 2-20, Table 2-21, and Table 2-22 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3AH.		

2.12.1 MSRs in the Intel® Xeon® Processor E5 v2 Product Family Based on Ivy Bridge-E Microarchitecture

Table 2-26 lists model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 v2 Product Family (based on Ivy Bridge-E microarchitecture). These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH; see Table 2-1. These processors supports the MSR interfaces listed in Table 2-20 and Table 2-26.

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 4EH, 78	IA32_PPIN_CTL (MSR_PPIN_CTL)	
Protected Processor Inventory Number Enable Control (R/W)		Package
0	LockOut (R/WO) See Table 2-2.	
1	Enable_PPIN (R/W) See Table 2-2.	
63:2	Reserved.	

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 4FH, 79	IA32_PPIN (MSR_PPIN)	
Protected Processor Inventory Number (R/O)		Package
63:0	Protected Processor Inventory Number (R/O) See Table 2-2.	
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.	Package
22:16	Reserved.	
23	PPIN_CAP (R/O) When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for a privileged system inventory agent to read PPIN from MSR_PPIN. When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.	Package
27:24	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.	Package
29	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.	Package
30	Programmable TJ OFFSET (R/O) When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify a temperature offset.	Package
39:31	Reserved.	
47:40	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	Package
63:48	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.		Core

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/WO) When set, locks bits 15:0 of this register until next reset.	
63:16	Reserved.	
Register Address: 179H, 377	IA32_MCG_CAP	
Global Machine Check Capability (R/O)		Thread
7:0	Count	
8	MCG_CTL_P	
9	MCG_EXT_P	
10	MCP_CMCI_P	
11	MCG_TES_P	
15:12	Reserved.	
23:16	MCG_EXT_CNT	
24	MCG_SER_P	
25	Reserved.	
26	MCG_ELOG_P	
63:27	Reserved.	
Register Address: 17FH, 383	MSR_ERROR_CONTROL	
MC Bank Error Configuration (R/W)		Package
0	Reserved.	
1	MemError Log Enable (R/W) When set, enables IMC status bank to log additional info in bits 36:32.	
63:2	Reserved.	

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target		Package
15:0	Reserved.	
23:16	Temperature Target (R/O) The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.	
27:24	TCC Activation Offset (R/W) Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only if MSR_PLATFORM_INFO.[30] is set.	
63:28	Reserved.	
Register Address: 1AEH, 430	MSR_TURBO_RATIO_LIMIT1	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 9C Maximum turbo ratio limit of 9 core active.	Package
15:8	Maximum Ratio Limit for 10C Maximum turbo ratio limit of 10 core active.	Package
23:16	Maximum Ratio Limit for 11C Maximum turbo ratio limit of 11 core active.	Package
31:24	Maximum Ratio Limit for 12C Maximum turbo ratio limit of 12 core active.	Package
63:32	Reserved.	
Register Address: 285H, 645	IA32_MC5_CTL2	
See Table 2-2.		Package
Register Address: 286H, 646	IA32_MC6_CTL2	
See Table 2-2.		Package
Register Address: 287H, 647	IA32_MC7_CTL2	
See Table 2-2.		Package
Register Address: 288H, 648	IA32_MC8_CTL2	
See Table 2-2.		Package
Register Address: 289H, 649	IA32_MC9_CTL2	
See Table 2-2.		Package
Register Address: 28AH, 650	IA32_MC10_CTL2	
See Table 2-2.		Package
Register Address: 28BH, 651	IA32_MC11_CTL2	
See Table 2-2.		Package
Register Address: 28CH, 652	IA32_MC12_CTL2	
See Table 2-2.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 28DH, 653	IA32_MC13_CTL2	
See Table 2-2.		Package
Register Address: 28EH, 654	IA32_MC14_CTL2	
See Table 2-2.		Package
Register Address: 28FH, 655	IA32_MC15_CTL2	
See Table 2-2.		Package
Register Address: 290H, 656	IA32_MC16_CTL2	
See Table 2-2.		Package
Register Address: 291H, 657	IA32_MC17_CTL2	
See Table 2-2.		Package
Register Address: 292H, 658	IA32_MC18_CTL2	
See Table 2-2.		Package
Register Address: 293H, 659	IA32_MC19_CTL2	
See Table 2-2.		Package
Register Address: 294H, 660	IA32_MC20_CTL2	
See Table 2-2.		Package
Register Address: 295H, 661	IA32_MC21_CTL2	
See Table 2-2.		Package
Register Address: 296H, 662	IA32_MC22_CTL2	
See Table 2-2.		Package
Register Address: 297H, 663	IA32_MC23_CTL2IA32_MC23_CTL2	
See Table 2-2.		Package
Register Address: 298H, 664	IA32_MC24_CTL2	
See Table 2-2.		Package
Register Address: 299H, 665	IA32_MC25_CTL2	
See Table 2-2.		Package
Register Address: 29AH, 666	IA32_MC26_CTL2	
See Table 2-2.		Package
Register Address: 29BH, 667	IA32_MC27_CTL2	
See Table 2-2.		Package
Register Address: 29CH, 668	IA32_MC28_CTL2	
See Table 2-2.		Package
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI module.		Package
Register Address: 415H, 1045	IA32_MC5_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI module.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from the Intel QPI module.		Package
Register Address: 417H, 1047	IA32_MC5_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from the Intel QPI module.		Package
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41BH, 1051	IA32_MC6_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41CH, 1052	IA32_MC7_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.		Package
Register Address: 41DH, 1053	IA32_MC7_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.		Package
Register Address: 41EH, 1054	IA32_MC7_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.		Package
Register Address: 41FH, 1055	IA32_MC7_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.		Package
Register Address: 420H, 1056	IA32_MC8_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.		Package
Register Address: 421H, 1057	IA32_MC8_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.		Package
Register Address: 422H, 1058	IA32_MC8_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC7 and MC 8 report MC errors from the two home agents.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 423H, 1059	IA32_MC8_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC7 and MC 8 report MC errors from the two home agents.		Package
Register Address: 424H, 1060	IA32_MC9_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 425H, 1061	IA32_MC9_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 426H, 1062	IA32_MC9_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 427H, 1063	IA32_MC9_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 428H, 1064	IA32_MC10_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 429H, 1065	IA32_MC10_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42AH, 1066	IA32_MC10_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42BH, 1067	IA32_MC10_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42CH, 1068	IA32_MC11_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.” Bank MC11 reports MC errors from a specific channel of the integrated memory controller.		Package
Register Address: 42DH, 1069	IA32_MC11_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.” Bank MC11 reports MC errors from a specific channel of the integrated memory controller.		Package
Register Address: 42EH, 1070	IA32_MC11_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.” Bank MC11 reports MC errors from a specific channel of the integrated memory controller.		Package
Register Address: 42FH, 1071	IA32_MC11_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.” Bank MC11 reports MC errors from a specific channel of the integrated memory controller.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 430H, 1072	IA32_MC12_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 431H, 1073	IA32_MC12_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 432H, 1074	IA32_MC12_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 433H, 1075	IA32_MC12_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 434H, 1076	IA32_MC13_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 435H, 1077	IA32_MC13_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 436H, 1078	IA32_MC13_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 437H, 1079	IA32_MC13_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 438H, 1080	IA32_MC14_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 439H, 1081	IA32_MC14_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43AH, 1082	IA32_MC14_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43BH, 1083	IA32_MC14_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43CH, 1084	IA32_MC15_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 43DH, 1085	IA32_MC15_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43EH, 1086	IA32_MC15_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43FH, 1087	IA32_MC15_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 440H, 1088	IA32_MC16_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 441H, 1089	IA32_MC16_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 442H, 1090	IA32_MC16_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 443H, 1091	IA32_MC16_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 444H, 1092	IA32_MC17_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 445H, 1093	IA32_MC17_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 446H, 1094	IA32_MC17_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 447H, 1095	IA32_MC17_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 448H, 1096	IA32_MC18_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 449H, 1097	IA32_MC18_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 44AH, 1098	IA32_MC18_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 44BH, 1099	IA32_MC18_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 44CH, 1100	IA32_MC19_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 44DH, 1101	IA32_MC19_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 44EH, 1102	IA32_MC19_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 44FH, 1103	IA32_MC19_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 450H, 1104	IA32_MC20_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs." Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 451H, 1105	IA32_MC20_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs." Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 452H, 1106	IA32_MC20_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs." Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 453H, 1107	IA32_MC20_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs." Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 454H, 1108	IA32_MC21_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 455H, 1109	IA32_MC21_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 456H, 1110	IA32_MC21_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 457H, 1111	IA32_MC21_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 458H, 1112	IA32_MC22_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 459H, 1113	IA32_MC22_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 45AH, 1114	IA32_MC22_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 45BH, 1115	IA32_MC22_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 45CH, 1116	IA32_MC23_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 45DH, 1117	IA32_MC23_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 45EH, 1118	IA32_MC23_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 45FH, 1119	IA32_MC23_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 460H, 1120	IA32_MC24_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 461H, 1121	IA32_MC24_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 462H, 1122	IA32_MC24_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 463H, 1123	IA32_MC24_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 464H, 1124	IA32_MC25_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC25 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 465H, 1125	IA32_MC25_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC25 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 466H, 1126	IA32_MC25_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC25 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 467H, 1127	IA32_MC2MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC25 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 468H, 1128	IA32_MC26_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC26 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 469H, 1129	IA32_MC26_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC26 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 46AH, 1130	IA32_MC26_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC26 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 46BH, 1131	IA32_MC26_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC26 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 46CH, 1132	IA32_MC27_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC27 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 46DH, 1133	IA32_MC27_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC27 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 46EH, 1134	IA32_MC27_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC27 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 46FH, 1135	IA32_MC27_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC27 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 470H, 1136	IA32_MC28_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC28 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 471H, 1137	IA32_MC28_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC28 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 472H, 1138	IA32_MC28_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC28 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 473H, 1139	IA32_MC28_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC28 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 613H, 1555	MSR_PKG_PERF_STATUS	
Package RAPL Perf Status (R/O)		Package
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
DRAM RAPL Power Limit Control (R/W) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	
DRAM RAPL Parameters (R/W) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, “PP0/PP1 RAPL Domains.”		Package
See Table 2-20, for other MSR definitions applicable to Intel Xeon processor E5 v2 with a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH.		

2.12.2 Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family

The Intel® Xeon® processor E7 v2 family (based on Ivy Bridge-E microarchitecture) with a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH supports the MSR interfaces listed in Table 2-20, Table 2-26, and Table 2-27.

Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	

Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Control Features in Intel 64 Processor (R/W) See Table 2-2.		Thread
0	Lock (R/WL)	
1	Enable VMX Inside SMX Operation (R/WL)	
2	Enable VMX Outside SMX Operation (R/WL)	
14:8	SENTER Local Functions Enables (R/WL)	
15	SENTER Global Functions Enable (R/WL)	
63:16	Reserved.	
Register Address: 179H, 377	IA32_MCG_CAP	
Global Machine Check Capability (R/O)		Thread
7:0	Count	
8	MCG_CTL_P	
9	MCG_EXT_P	
10	MCP_CMCI_P	
11	MCG_TES_P	
15:12	Reserved.	
23:16	MCG_EXT_CNT	
24	MCG_SER_P	
63:25	Reserved.	
Register Address: 17AH, 378	IA32_MCG_STATUS	
Global Machine Check Status (R/W)		Thread
0	RIPV	
1	EIPV	
2	MCIP	
3	LMCE Signaled	
63:4	Reserved.	
Register Address: 1AEH, 430	MSR_TURBO_RATIO_LIMIT1	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 9C Maximum turbo ratio limit of 9 core active.	Package
15:8	Maximum Ratio Limit for 10C Maximum turbo ratio limit of 10core active.	Package
23:16	Maximum Ratio Limit for 11C Maximum turbo ratio limit of 11 core active.	Package
31:24	Maximum Ratio Limit for 12C Maximum turbo ratio limit of 12 core active.	Package

Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
39:32	Maximum Ratio Limit for 13C Maximum turbo ratio limit of 13 core active.	Package
47:40	Maximum Ratio Limit for 14C Maximum turbo ratio limit of 14 core active.	Package
55:48	Maximum Ratio Limit for 15C Maximum turbo ratio limit of 15 core active.	Package
62:56	Reserved.	
63	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).	Package
Register Address: 29DH, 669	IA32_MC29_CTL2	
See Table 2-2.		Package
Register Address: 29EH, 670	IA32_MC30_CTL2	
See Table 2-2.		Package
Register Address: 29FH, 671	IA32_MC31_CTL2	
See Table 2-2.		Package
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Section 21.3.1.1.1, "Processor Event Based Sampling (PEBS)."		Thread
n:0	Enable PEBS on IA32_PMCx. (R/W)	
31:*n*+1	Reserved.	
32+*m*:32	Enable Load Latency on IA32_PMCx. (R/W)	
63:33+*m*	Reserved.	
Register Address: 41BH, 1051	IA32_MC6_MISC	
Misc MAC Information of Integrated I/O (R/O) See Section 17.3.2.4.		Package
5:0	Recoverable Address LSB	
8:6	Address Mode	
15:9	Reserved.	
31:16	PCI Express Requestor ID	
39:32	PCI Express Segment Number	
63:32	Reserved.	
Register Address: 474H, 1140	IA32_MC29_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 475H, 1141	IA32_MC29_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package

Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 476H, 1142	IA32_MC29_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 477H, 1143	IA32_MC29_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 478H, 1144	IA32_MC30_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 479H, 1145	IA32_MC30_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 47AH, 1146	IA32_MC30_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 47BH, 1147	IA32_MC30_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 47CH, 1148	IA32_MC31_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC31 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 47DH, 1149	IA32_MC31_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC31 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 47EH, 1150	IA32_MC31_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC31 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
Register Address: 47FH, 1147	IA32_MC31_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC31 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.		Package
See Table 2-20, Table 2-26 for other MSR definitions applicable to Intel Xeon processor E7 v2 with a CPUID Signature DisplayFamily_DisplayModel value of 06_3AH.		

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.12.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Intel Xeon Processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-24 and Table 2-28. For complete detail of the uncore PMU, refer to Intel

Xeon Processor E5 v2 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH.

Table 2-28. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: C00H, 3072	MSR_PMON_GLOBAL_CTL	
Uncore PerfMon Per-Socket Global Control		Package
Register Address: C01H, 3073	MSR_PMON_GLOBAL_STATUS	
Uncore PerfMon Per-Socket Global Status		Package
Register Address: C06H, 3078	MSR_PMON_GLOBAL_CONFIG	
Uncore PerfMon Per-Socket Global Configuration		Package
Register Address: C15H, 3093	MSR_U_PMON_BOX_STATUS	
Uncore U-box PerfMon U-Box Wide Status		Package
Register Address: C35H, 3125	MSR_PCU_PMON_BOX_STATUS	
Uncore PCU PerfMon Box Wide Status		Package
Register Address: D1AH, 3354	MSR_C0_PMON_BOX_FILTER1	
Uncore C-Box 0 PerfMon Box Wide Filter1		Package
Register Address: D3AH, 3386	MSR_C1_PMON_BOX_FILTER1	
Uncore C-Box 1 PerfMon Box Wide Filter1		Package
Register Address: D5AH, 3418	MSR_C2_PMON_BOX_FILTER1	
Uncore C-Box 2 PerfMon Box Wide Filter1		Package
Register Address: D7AH, 3450	MSR_C3_PMON_BOX_FILTER1	
Uncore C-Box 3 PerfMon Box Wide Filter1		Package
Register Address: D9AH, 3482	MSR_C4_PMON_BOX_FILTER1	
Uncore C-Box 4 PerfMon Box Wide Filter1		Package
Register Address: DBAH, 3514	MSR_C5_PMON_BOX_FILTER1	
Uncore C-Box 5 PerfMon Box Wide Filter1		Package
Register Address: DDAH, 3546	MSR_C6_PMON_BOX_FILTER1	
Uncore C-Box 6 PerfMon Box Wide Filter1		Package
Register Address: DFAH, 3578	MSR_C7_PMON_BOX_FILTER1	
Uncore C-Box 7 PerfMon Box Wide Filter1		Package
Register Address: E04H, 3588	MSR_C8_PMON_BOX_CTL	
Uncore C-Box 8 PerfMon Local Box Wide Control		Package
Register Address: E10H, 3600	MSR_C8_PMON_EVNTSEL0	
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 0		Package
Register Address: E11H, 3601	MSR_C8_PMON_EVNTSEL1	
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 1		Package
Register Address: E12H, 3602	MSR_C8_PMON_EVNTSEL2	
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 2		Package
Register Address: E13H, 3603	MSR_C8_PMON_EVNTSEL3	

Table 2-28. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 3		Package
Register Address: E14H, 3604	MSR_C8_PMON_BOX_FILTER	
Uncore C-Box 8 PerfMon Box Wide Filter		Package
Register Address: E16H, 3606	MSR_C8_PMON_CTR0	
Uncore C-Box 8 PerfMon Counter 0		Package
Register Address: E17H, 3607	MSR_C8_PMON_CTR1	
Uncore C-Box 8 PerfMon Counter 1		Package
Register Address: E18H, 3608	MSR_C8_PMON_CTR2	
Uncore C-Box 8 PerfMon Counter 2		Package
Register Address: E19H, 3609	MSR_C8_PMON_CTR3	
Uncore C-Box 8 PerfMon Counter 3		Package
Register Address: E1AH, 3610	MSR_C8_PMON_BOX_FILTER1	
Uncore C-Box 8 PerfMon Box Wide Filter1		Package
Register Address: E24H, 3620	MSR_C9_PMON_BOX_CTL	
Uncore C-Box 9 PerfMon Local Box Wide Control		Package
Register Address: E30H, 3632	MSR_C9_PMON_EVNTSEL0	
Uncore C-Box 9 PerfMon Event Select for C-box 9 Counter 0		Package
Register Address: E31H, 3633	MSR_C9_PMON_EVNTSEL1	
Uncore C-Box 9 PerfMon Event Select for C-box 9 Counter 1		Package
Register Address: E32H, 3634	MSR_C9_PMON_EVNTSEL2	
Uncore C-Box 9 PerfMon Event Select for C-box 9 Counter 2		Package
Register Address: E33H, 3635	MSR_C9_PMON_EVNTSEL3	
Uncore C-Box 9 PerfMon Event Select for C-box 9 Counter 3		Package
Register Address: E34H, 3636	MSR_C9_PMON_BOX_FILTER	
Uncore C-Box 9 PerfMon Box Wide Filter		Package
Register Address: E36H, 3638	MSR_C9_PMON_CTR0	
Uncore C-Box 9 PerfMon Counter 0		Package
Register Address: E37H, 3639	MSR_C9_PMON_CTR1	
Uncore C-Box 9 PerfMon Counter 1		Package
Register Address: E38H, 3640	MSR_C9_PMON_CTR2	
Uncore C-Box 9 PerfMon Counter 2		Package
Register Address: E39H, 3641	MSR_C9_PMON_CTR3	
Uncore C-Box 9 PerfMon Counter 3		Package
Register Address: E3AH, 3642	MSR_C9_PMON_BOX_FILTER1	
Uncore C-Box 9 PerfMon Box Wide Filter1		Package
Register Address: E44H, 3652	MSR_C10_PMON_BOX_CTL	
Uncore C-Box 10 PerfMon Local Box Wide Control		Package

Table 2-28. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E50H, 3664	MSR_C10_PMON_EVNTSEL0	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 0		Package
Register Address: E51H, 3665	MSR_C10_PMON_EVNTSEL1	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 1		Package
Register Address: E52H, 3666	MSR_C10_PMON_EVNTSEL2	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 2		Package
Register Address: E53H, 3667	MSR_C10_PMON_EVNTSEL3	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 3		Package
Register Address: E54H, 3668	MSR_C10_PMON_BOX_FILTER	
Uncore C-Box 10 PerfMon Box Wide Filter		Package
Register Address: E56H, 3670	MSR_C10_PMON_CTR0	
Uncore C-Box 10 PerfMon Counter 0		Package
Register Address: E57H, 3671	MSR_C10_PMON_CTR1	
Uncore C-Box 10 PerfMon Counter 1		Package
Register Address: E58H, 3672	MSR_C10_PMON_CTR2	
Uncore C-Box 10 PerfMon Counter 2		Package
Register Address: E59H, 3673	MSR_C10_PMON_CTR3	
Uncore C-Box 10 PerfMon Counter 3		Package
Register Address: E5AH, 3674	MSR_C10_PMON_BOX_FILTER1	
Uncore C-Box 10 PerfMon Box Wide Filter1		Package
Register Address: E64H, 3684	MSR_C11_PMON_BOX_CTL	
Uncore C-Box 11 PerfMon Local Box Wide Control		Package
Register Address: E70H, 3696	MSR_C11_PMON_EVNTSEL0	
Uncore C-Box 11 PerfMon Event Select for C-Box 11 Counter 0		Package
Register Address: E71H, 3697	MSR_C11_PMON_EVNTSEL1	
Uncore C-Box 11 PerfMon Event Select for C-Box 11 Counter 1		Package
Register Address: E72H, 3698	MSR_C11_PMON_EVNTSEL2	
Uncore C-Box 11 PerfMon Event Select for C-Box 11 Counter 2		Package
Register Address: E73H, 3699	MSR_C11_PMON_EVNTSEL3	
Uncore C-Box 11 PerfMon Event Select for C-Box 11 Counter 3		Package
Register Address: E74H, 3700	MSR_C11_PMON_BOX_FILTER	
Uncore C-Box 11 PerfMon Box Wide Filter		Package
Register Address: E76H, 3702	MSR_C11_PMON_CTR0	
Uncore C-Box 11 PerfMon Counter 0		Package
Register Address: E77H, 3703	MSR_C11_PMON_CTR1	
Uncore C-Box 11 PerfMon Counter 1		Package
Register Address: E78H, 3704	MSR_C11_PMON_CTR2	

Table 2-28. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 11 PerfMon Counter 2		Package
Register Address: E79H, 3705	MSR_C11_PMON_CTR3	
Uncore C-Box 11 PerfMon Counter 3		Package
Register Address: E7AH, 3706	MSR_C11_PMON_BOX_FILTER1	
Uncore C-Box 11 PerfMon Box Wide Filter1		Package
Register Address: E84H, 3716	MSR_C12_PMON_BOX_CTL	
Uncore C-Box 12 PerfMon Local Box Wide Control		Package
Register Address: E90H, 3728	MSR_C12_PMON_EVNTSEL0	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 0		Package
Register Address: E91H, 3729	MSR_C12_PMON_EVNTSEL1	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 1		Package
Register Address: E92H, 3730	MSR_C12_PMON_EVNTSEL2	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 2		Package
Register Address: E93H, 3731	MSR_C12_PMON_EVNTSEL3	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 3		Package
Register Address: E94H, 3732	MSR_C12_PMON_BOX_FILTER	
Uncore C-Box 12 PerfMon Box Wide Filter		Package
Register Address: E96H, 3734	MSR_C12_PMON_CTR0	
Uncore C-Box 12 PerfMon Counter 0		Package
Register Address: E97H, 3735	MSR_C12_PMON_CTR1	
Uncore C-Box 12 PerfMon Counter 1		Package
Register Address: E98H, 3736	MSR_C12_PMON_CTR2	
Uncore C-Box 12 PerfMon Counter 2		Package
Register Address: E99H, 3737	MSR_C12_PMON_CTR3	
Uncore C-Box 12 PerfMon Counter 3		Package
Register Address: E9AH, 3738	MSR_C12_PMON_BOX_FILTER1	
Uncore C-Box 12 PerfMon Box Wide Filter1		Package
Register Address: EA4H, 3748	MSR_C13_PMON_BOX_CTL	
Uncore C-Box 13 PerfMon Local Box Wide Control		Package
Register Address: EB0H, 3760	MSR_C13_PMON_EVNTSEL0	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 0		Package
Register Address: EB1H, 3761	MSR_C13_PMON_EVNTSEL1	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 1		Package
Register Address: EB2H, 3762	MSR_C13_PMON_EVNTSEL2	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 2		Package
Register Address: EB3H, 3763	MSR_C13_PMON_EVNTSEL3	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 3		Package

Table 2-28. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: EB4H, 3764	MSR_C13_PMON_BOX_FILTER	
Uncore C-Box 13 PerfMon Box Wide Filter		Package
Register Address: EB6H, 3766	MSR_C13_PMON_CTR0	
Uncore C-Box 13 PerfMon Counter 0		Package
Register Address: EB7H, 3767	MSR_C13_PMON_CTR1	
Uncore C-Box 13 PerfMon Counter 1		Package
Register Address: EB8H, 3768	MSR_C13_PMON_CTR2	
Uncore C-Box 13 PerfMon Counter 2		Package
Register Address: EB9H, 3769	MSR_C13_PMON_CTR3	
Uncore C-Box 13 PerfMon Counter 3		Package
Register Address: EBAH, 3770	MSR_C13_PMON_BOX_FILTER1	
Uncore C-Box 13 PerfMon Box Wide Filter1		Package
Register Address: EC4H, 3780	MSR_C14_PMON_BOX_CTL	
Uncore C-Box 14 PerfMon Local Box Wide Control		Package
Register Address: ED0H, 3792	MSR_C14_PMON_EVNTSEL0	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 0		Package
Register Address: ED1H, 3793	MSR_C14_PMON_EVNTSEL1	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 1		Package
Register Address: ED2H, 3794	MSR_C14_PMON_EVNTSEL2	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 2		Package
Register Address: ED3H, 3795	MSR_C14_PMON_EVNTSEL3	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 3		Package
Register Address: ED4H, 3796	MSR_C14_PMON_BOX_FILTER	
Uncore C-Box 14 PerfMon Box Wide Filter		Package
Register Address: ED6H, 3798	MSR_C14_PMON_CTR0	
Uncore C-Box 14 PerfMon Counter 0		Package
Register Address: ED7H, 3799	MSR_C14_PMON_CTR1	
Uncore C-Box 14 PerfMon Counter 1		Package
Register Address: ED8H, 3800	MSR_C14_PMON_CTR2	
Uncore C-Box 14 PerfMon Counter 2		Package
Register Address: ED9H, 3801	MSR_C14_PMON_CTR3	
Uncore C-Box 14 PerfMon Counter 3		Package
Register Address: EDAH, 3802	MSR_C14_PMON_BOX_FILTER1	
Uncore C-Box 14 PerfMon Box Wide Filter1		Package

2.13 MSRS IN THE 4TH GENERATION INTEL® CORE™ PROCESSORS BASED ON HASWELL MICROARCHITECTURE

The 4th generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v3 product family (based on Haswell microarchitecture), with a CPUID Signature DisplayFamily_DisplayModel value of 06_3CH, 06_45H, or 06_46H, support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, and Table 2-29. For an MSR listed in Table 2-20 that also appears in Table 2-29, Table 2-29 supersedes Table 2-20.

The MSRs listed in Table 2-29 also apply to processors based on Haswell-E microarchitecture (see Section 2.14).

Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3BH, 59	IA32_TSC_ADJUST	
Per-Logical-Processor TSC ADJUST (R/W) See Table 2-2.		Thread
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.	Package
27:16	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.	Package
29	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.	Package
31:30	Reserved.	
32	Low Power Mode Support (LPM) (R/O) When set to 1, indicates that LPM is supported. When set to 0, indicates LPM is not supported.	Package
34:33	Number of ConfigTDP Levels (R/O) 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 03: Reserved.	Package
39:35	Reserved.	
47:40	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	Package
55:48	Minimum Operating Ratio (R/O) Contains the minimum supported operating ratio in units of 100 MHz.	Package
63:56	Reserved.	

Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 186H, 390	IA32_PERFEVTSEL0	
Performance Event Select for Counter 0 (R/W) Supports all fields described inTable 2-2 and the fields below.		Thread
32	IN_TX: See Section 21.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.	
Register Address: 187H, 391	IA32_PERFEVTSEL1	
Performance Event Select for Counter 1 (R/W) Supports all fields described inTable 2-2 and the fields below.		Thread
32	IN_TX: See Section 21.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.	
Register Address: 188H, 392	IA32_PERFEVTSEL2	
Performance Event Select for Counter 2 (R/W) Supports all fields described inTable 2-2 and the fields below.		Thread
32	IN_TX: See Section 21.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.	
33	IN_TXCP: See Section 21.3.6.5.1. When IN_TXCP=1 & IN_TX=1 and in sampling, a spurious PMI may occur and transactions may continuously abort near overflow conditions. Software should favor using IN_TXCP for counting over sampling. If sampling, software should use large "sample-after" value after clearing the counter configured to use IN_TXCP and also always reset the counter even when no overflow condition was reported.	
Register Address: 189H, 393	IA32_PERFEVTSEL3	
Performance Event Select for Counter 3 (R/W) Supports all fields described inTable 2-2 and the fields below.		Thread
32	IN_TX: See Section 21.3.6.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.	
Register Address: 1C8H, 456	MSR_LBR_SELECT	
Last Branch Record Filtering Select Register (R/W)		Thread
0	CPL_EQ_0	
1	CPL_NEQ_0	
2	JCC	
3	NEAR_REL_CALL	
4	NEAR_IND_CALL	
5	NEAR_RET	
6	NEAR_IND_JMP	
7	NEAR_REL_JMP	

Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
8	FAR_BRANCH	
9	EN_CALL_STACK	
63:9	Reserved.	
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W) See Table 2-2.		Thread
0	LBR: Last Branch Record	
1	BTF	
5:2	Reserved.	
6	TR: Branch Trace	
7	BTS: Log Branch Trace Message to BTS Buffer	
8	BTINT	
9	BTS_OFF_OS	
10	BTS_OFF_USER	
11	FREEZE_LBR_ON_PMI	
12	FREEZE_PERFMON_ON_PMI	
13	ENABLE_UNCORE_PMI	
14	FREEZE_WHILE_SMM	
15	RTM_DEBUG	
63:15	Reserved.	
Register Address: 491H, 1169	IA32_VMX_VMFUNC	
Capability Reporting Register of VM-Function Controls (R/O) See Table 2-2.		Thread
Register Address: 60BH, 1548	MSR_PKGC_IRTL1	
Package C6/C7 Interrupt Response Limit 1 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7 state. The latency programmed in this register is for the shorter-latency sub C-states used by an MWAIT hint to a C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
9:0	Interrupt Response Time Limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.	
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	

Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:16	Reserved.	
Register Address: 60CH, 1548	MSR_PKGC_IRTL2	
Package C6/C7 Interrupt Response Limit 2 (R/W) This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7 state. The latency programmed in this register is for the longer-latency sub C-states used by an MWAIT hint to a C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
9:0	Interrupt response time limit (R/W) Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.	
12:10	Time Unit (R/W) Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.	
14:13	Reserved.	
15	Valid (R/W) Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.	
63:16	Reserved.	
Register Address: 613H, 1555	MSR_PKG_PERF_STATUS	
PKG Perf Status (R/O) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 648H, 1608	MSR_CONFIG_TDP_NOMINAL	
Base TDP Ratio (R/O)		Package
7:0	Config_TDP_Base Base TDP level ratio to be used for this specific processor (in units of 100 MHz).	
63:8	Reserved.	
Register Address: 649H, 1609	MSR_CONFIG_TDP_LEVEL1	
ConfigTDP Level 1 Ratio and Power Level (R/O)		Package
14:0	PKG_TDP_LVL1 Power setting for ConfigTDP Level 1.	
15	Reserved.	
23:16	Config_TDP_LVL1_Ratio ConfigTDP level 1 ratio to be used for this specific processor.	

Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:24	Reserved.	
46:32	PKG_MAX_PWR_LVL1 Max Power setting allowed for ConfigTDP Level 1.	
62:47	PKG_MIN_PWR_LVL1 MIN Power setting allowed for ConfigTDP Level 1.	
63	Reserved.	
Register Address: 64AH, 1610	MSR_CONFIG_TDP_LEVEL2	
ConfigTDP Level 2 Ratio and Power Level (R/O)		Package
14:0	PKG_TDP_LVL2 Power setting for ConfigTDP Level 2.	
15	Reserved.	
23:16	Config_TDP_LVL2_Ratio ConfigTDP level 2 ratio to be used for this specific processor.	
31:24	Reserved.	
46:32	PKG_MAX_PWR_LVL2 Max Power setting allowed for ConfigTDP Level 2.	
62:47	PKG_MIN_PWR_LVL2 MIN Power setting allowed for ConfigTDP Level 2.	
63	Reserved.	
Register Address: 64BH, 1611	MSR_CONFIG_TDP_CONTROL	
ConfigTDP Control (R/W)		Package
1:0	TDP_LEVEL (RW/L) System BIOS can program this field.	
30:2	Reserved.	
31	Config_TDP_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.	
63:32	Reserved.	
Register Address: 64CH, 1612	MSR_TURBO_ACTIVATION_RATIO	
ConfigTDP Control (R/W)		Package
7:0	MAX_NON_TURBO_RATIO (RW/L) System BIOS can program this field.	
30:8	Reserved.	
31	TURBO_ACTIVATION_RATIO_Lock (RW/L) When this bit is set, the content of this register is locked until a reset.	
63:32	Reserved.	
Register Address: C80H, 3200	IA32_DEBUG_INTERFACE	
Silicon Debug Feature Control (R/W) See Table 2-2.		Package

2.13.1 MSRs in the 4th Generation Intel® Core™ Processor Family Based on Haswell Microarchitecture

Table 2-30 lists model-specific registers (MSRs) that are specific to the 4th generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200 v3 product family (based on Haswell microarchitecture). These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3CH, 06_45H, or 06_46H; see Table 2-1.

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.		Core
3:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s Package C states C7 are not available to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3CH.	
9:4	Reserved.	
10	I/O MWAIT Redirection Enable (R/W)	
14:11	Reserved	
15	CFG Lock (R/WO)	
24:16	Reserved.	
25	C3 State Auto Demotion Enable (R/W)	
26	C1 State Auto Demotion Enable (R/W)	
27	Enable C3 Undemotion (R/W)	
28	Enable C1 Undemotion (R/W)	
63:29	Reserved.	
Register Address: 17DH, 381	MSR_SMM_MCA_CAP	
Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.		Thread
57:0	Reserved.	
58	SMM_Code_Access_Chk (SMM-RO) If set to 1, indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
59	Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.	
63:60	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
63:32	Reserved.	
Register Address: 391H, 913	MSR_UNC_PERF_GLOBAL_CTRL	
Uncore PMU Global Control		Package
0	Core 0 select.	
1	Core 1 select.	
2	Core 2 select.	
3	Core 3 select.	
18:4	Reserved.	
29	Enable all uncore counters.	
30	Enable wake on PMI.	
31	Enable Freezing counter when overflow.	
63:32	Reserved.	
Register Address: 392H, 914	MSR_UNC_PERF_GLOBAL_STATUS	
Uncore PMU Main Status		Package
0	Fixed counter overflowed.	
1	An ARB counter overflowed.	
2	Reserved.	
3	A CBox counter overflowed (on any slice).	
63:4	Reserved.	
Register Address: 394H, 916	MSR_UNC_PERF_FIXED_CTRL	
Uncore Fixed Counter Control (R/W)		Package
19:0	Reserved.	
20	Enable overflow propagation.	
21	Reserved.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
22	Enable counting.	
63:23	Reserved.	
Register Address: 395H, 917	MSR_UNC_PERF_FIXED_CTR	
Uncore Fixed Counter		Package
47:0	Current count.	
63:48	Reserved.	
Register Address: 396H, 918	MSR_UNC_CBO_CONFIG	
Uncore C-Box Configuration Information (R/O)		Package
3:0	Encoded number of C-Box, derive value by “-1“.	
63:4	Reserved.	
Register Address: 3B0H, 946	MSR_UNC_ARB_PERFCTR0	
Uncore Arb Unit, Performance Counter 0		Package
Register Address: 3B1H, 947	MSR_UNC_ARB_PERFCTR1	
Uncore Arb Unit, Performance Counter 1		Package
Register Address: 3B2H, 944	MSR_UNC_ARB_PERFEVTSEL0	
Uncore Arb Unit, Counter 0 Event Select MSR		Package
Register Address: 3B3H, 945	MSR_UNC_ARB_PERFEVTSEL1	
Uncore Arb Unit, Counter 1 Event Select MSR		Package
Register Address: 4E0H, 1248	MSR_SMM_FEATURE_CONTROL	
Enhanced SMM Feature Control (SMM-RW) Reports SMM capability Enhancement. Accessible only while in SMM.		Package
0	Lock (SMM-RWO) When set to ‘1’ locks this register from further changes.	
1	Reserved.	
2	SMM_Code_Chk_En (SMM-RW) This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to ‘0’ (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to ‘1’ any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.	
63:3	Reserved.	
Register Address: 4E2H, 1250	MSR_SMM_DELAYED	
SMM Delayed (SMM-RO) Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.		Package

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
N-1:0	LOG_PROC_STATE (SMM-RO) Each bit represents a logical processor of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.	
63:N	Reserved.	
Register Address: 4E3H, 1251	MSR_SMM_BLOCKED	
SMM Blocked (SMM-RO) Reports the blocked state of all logical processors in the package. Available only while in SMM.		Package
N-1:0	LOG_PROC_STATE (SMM-RO) Each bit represents a logical processor of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.	
63:N	Reserved.	
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers Used in RAPL Interfaces (R/O)		Package
3:0	Power Units See Section 16.10.1, "RAPL Interfaces."	Package
7:4	Reserved.	Package
12:8	Energy Status Units Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).	Package
15:13	Reserved.	Package
19:16	Time Units See Section 16.10.1, "RAPL Interfaces."	Package
63:20	Reserved.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 640H, 1600	MSR_PP1_POWER_LIMIT	
PP1 RAPL Power Limit Control (R/W) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 641H, 1601	MSR_PP1_ENERGY_STATUS	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
PP1 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 642H, 1602	MSR_PP1_POLICY	
PP1 Balance Policy (R/W) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 690H, 1680	MSR_CORE_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)		Package
0	PROCHOT Status (R0) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.	
3:2	Reserved.	
4	Graphics Driver Status (R0) When set, frequency is reduced below the operating system request due to Processor Graphics driver override.	
5	Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.	
6	VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.	
7	Reserved.	
8	Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).	
9	Core Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to domain-level power limiting.	
10	Package-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.	
11	Package-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.	
12	Max Turbo Limit Status (R0) When set, frequency is reduced below the operating system request due to multi-core turbo limits.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
13	Turbo Transition Attenuation Status (R0) When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.	
15:14	Reserved.	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19:18	Reserved.	
20	Graphics Driver Log When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
21	Autonomous Utilization-Based Frequency Control Log When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	Reserved.	
24	Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Core Power Limiting Log When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
26	Package-Level PL1 Power Limiting Log When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
27	Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
29	Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:30	Reserved.	
Register Address: 6B0H, 1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in the Processor Graphics (R/W) (Frequency refers to processor graphics frequency.)		Package
0	PROCHOT Status (R0) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.	
3:2	Reserved.	
4	Graphics Driver Status (R0) When set, frequency is reduced below the operating system request due to Processor Graphics driver override.	
5	Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.	
6	VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.	
7	Reserved.	
8	Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).	
9	Graphics Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to domain-level power limiting.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
10	Package-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.	
11	Package-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.	
15:12	Reserved.	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19:18	Reserved.	
20	Graphics Driver Log When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
21	Autonomous Utilization-Based Frequency Control Log When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	Reserved.	
24	Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Core Power Limiting Log When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
26	Package-Level PL1 Power Limiting Log When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
27	Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
29	Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:30	Reserved.	
Register Address: 6B1H, 1713	MSR_RING_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in the Ring Interconnect (R/W) (Frequency refers to ring interconnect in the uncore.)		Package
0	PROCHOT Status (R0) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.	
5:2	Reserved.	
6	VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.	
7	Reserved.	
8	Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).	
9	Reserved.	
10	Package-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL1.	
11	Package-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package-level power limiting PL2.	
15:12	Reserved.	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19:18	Reserved.	
20	Graphics Driver Log When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
21	Autonomous Utilization-Based Frequency Control Log When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	Reserved.	
24	Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Core Power Limiting Log When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
26	Package-Level PL1 Power Limiting Log When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
27	Package-Level PL2 Power Limiting Log When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
29	Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:30	Reserved.	
Register Address: 700H, 1792	MSR_UNC_CBO_0_PERFEVTSEL0	
Uncore C-Box 0, Counter 0 Event Select MSR		Package
Register Address: 701H, 1793	MSR_UNC_CBO_0_PERFEVTSEL1	
Uncore C-Box 0, Counter 1 Event Select MSR		Package
Register Address: 706H, 1798	MSR_UNC_CBO_0_PERFCTR0	
Uncore C-Box 0, Performance Counter 0		Package
Register Address: 707H, 1799	MSR_UNC_CBO_0_PERFCTR1	
Uncore C-Box 0, Performance Counter 1		Package
Register Address: 710H, 1808	MSR_UNC_CBO_1_PERFEVTSEL0	
Uncore C-Box 1, Counter 0 Event Select MSR		Package
Register Address: 711H, 1809	MSR_UNC_CBO_1_PERFEVTSEL1	
Uncore C-Box 1, Counter 1 Event Select MSR		Package
Register Address: 716H, 1814	MSR_UNC_CBO_1_PERFCTR0	
Uncore C-Box 1, Performance Counter 0		Package
Register Address: 717H, 1815	MSR_UNC_CBO_1_PERFCTR1	
Uncore C-Box 1, Performance Counter 1		Package
Register Address: 720H, 1824	MSR_UNC_CBO_2_PERFEVTSEL0	
Uncore C-Box 2, Counter 0 Event Select MSR		Package
Register Address: 721H, 1824	MSR_UNC_CBO_2_PERFEVTSEL1	
Uncore C-Box 2, Counter 1 Event Select MSR		Package
Register Address: 726H, 1830	MSR_UNC_CBO_2_PERFCTR0	
Uncore C-Box 2, Performance Counter 0		Package
Register Address: 727H, 1831	MSR_UNC_CBO_2_PERFCTR1	
Uncore C-Box 2, Performance Counter 1		Package
Register Address: 730H, 1840	MSR_UNC_CBO_3_PERFEVTSEL0	
Uncore C-Box 3, Counter 0 Event Select MSR		Package
Register Address: 731H, 1841	MSR_UNC_CBO_3_PERFEVTSEL1	
Uncore C-Box 3, Counter 1 Event Select MSR		Package
Register Address: 736H, 1846	MSR_UNC_CBO_3_PERFCTR0	
Uncore C-Box 3, Performance Counter 0		Package
Register Address: 737H, 1847	MSR_UNC_CBO_3_PERFCTR1	
Uncore C-Box 3, Performance Counter 1		Package
See Table 2-20, Table 2-21, Table 2-22, Table 2-25, and Table 2-29 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 063CH or 06_46H.		

2.13.2 Additional Residency MSRs Supported in 4th Generation Intel® Core™ Processors

The 4th generation Intel® Core™ processor family (based on Haswell microarchitecture) with a CPUID Signature DisplayFamily_DisplayModel value of 06_45H supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-30, and Table 2-31.

Table 2-31. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_45H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.		Core
3:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10	
9:4	Reserved.	
10	I/O MWAIT Redirection Enable (R/W)	
14:11	Reserved.	
15	CFG Lock (R/WO)	
24:16	Reserved.	
25	C3 State Auto Demotion Enable (R/W)	
26	C1 State Auto Demotion Enable (R/W)	
27	Enable C3 Undemotion (R/W)	
28	Enable C1 Undemotion (R/W)	
63:29	Reserved.	
Register Address: 630H, 1584	MSR_PKG_C8_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
59:0	Package C8 Residency Counter (R/O) Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.	
63:60	Reserved.	
Register Address: 631H, 1585	MSR_PKG_C9_RESIDENCY	

Table 2-31. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_45H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
59:0	Package C9 Residency Counter (R/O) Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.	
63:60	Reserved.	
Register Address: 632H, 1586	MSR_PKG_C10_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
59:0	Package C10 Residency Counter (R/O) Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.	
63:60	Reserved.	
See Table 2-20, Table 2-21, Table 2-22, Table 2-29, and Table 2-30 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_45H.		

2.14 MSRS IN THE INTEL® XEON® PROCESSOR E5 V3 AND E7 V3 PRODUCT FAMILY

The Intel® Xeon® processor E5 v3 family and the Intel® Xeon® processor E7 v3 family are based on Haswell-E microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_3F). These processors support the MSR interfaces listed in Table 2-20, Table 2-29, and Table 2-32.

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 35H, 53	MSR_CORE_THREAD_COUNT	
Configured State of Enabled Processor Core Count and Logical Processor Count (R/O) ▪ After a Power-On RESET, enumerates factory configuration of the number of processor cores and logical processors in the physical package. ▪ Following the sequence of (i) BIOS modified a Configuration Mask which selects a subset of processor cores to be active post RESET and (ii) a RESET event after the modification, enumerates the current configuration of enabled processor core count and logical processor count in the physical package.		Package
15:0	THREAD_COUNT (R/O) The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.	
31:16	Core_COUNT (R/O) The number of processor cores that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.	
63:32	Reserved.	
Register Address: 53H, 83	MSR_THREAD_ID_INFO	
A Hardware Assigned ID for the Logical Processor (R/O)		Thread

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
7:0	Logical_Processor_ID (R/O) An implementation-specific numerical value physically assigned to each logical processor. This ID is not related to Initial APIC ID or x2APIC ID, it is unique within a physical package.	
63:8	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.		Core
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W)	
14:11	Reserved.	
15	CFG Lock (R/WO)	
24:16	Reserved.	
25	C3 State Auto Demotion Enable (R/W)	
26	C1 State Auto Demotion Enable (R/W)	
27	Enable C3 Undemotion (R/W)	
28	Enable C1 Undemotion (R/W)	
29	Package C State Demotion Enable (R/W)	
30	Package C State Undemotion Enable (R/W)	
63:31	Reserved.	
Register Address: 179H, 377	IA32_MCG_CAP	
Global Machine Check Capability (R/O)		Thread
7:0	Count	
8	MCG_CTL_P	
9	MCG_EXT_P	
10	MCP_CMCI_P	
11	MCG_TES_P	
15:12	Reserved.	

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
23:16	MCG_EXT_CNT	
24	MCG_SER_P	
25	MCG_EM_P	
26	MCG_ELOG_P	
63:27	Reserved.	
Register Address: 17DH, 381	MSR_SMM_MCA_CAP	
Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.		Thread
57:0	Reserved.	
58	SMM_Code_Access_Chk (SMM-RO) If set to 1, indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.	
59	Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.	
63:60	Reserved.	
Register Address: 17FH, 383	MSR_ERROR_CONTROL	
MC Bank Error Configuration (R/W)		Package
0	Reserved.	
1	MemError Log Enable (R/W) When set, enables IMC status bank to log additional info in bits 36:32.	
63:2	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
39:32	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5 core active.	Package
47:40	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6 core active.	Package
55:48	Maximum Ratio Limit for 7C Maximum turbo ratio limit of 7 core active.	Package

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:56	Maximum Ratio Limit for 8C Maximum turbo ratio limit of 8 core active.	Package
Register Address: 1AEH, 430	MSR_TURBO_RATIO_LIMIT1	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 9C Maximum turbo ratio limit of 9 core active.	Package
15:8	Maximum Ratio Limit for 10C Maximum turbo ratio limit of 10 core active.	Package
23:16	Maximum Ratio Limit for 11C Maximum turbo ratio limit of 11 core active.	Package
31:24	Maximum Ratio Limit for 12C Maximum turbo ratio limit of 12 core active.	Package
39:32	Maximum Ratio Limit for 13C Maximum turbo ratio limit of 13 core active.	Package
47:40	Maximum Ratio Limit for 14C Maximum turbo ratio limit of 14 core active.	Package
55:48	Maximum Ratio Limit for 15C Maximum turbo ratio limit of 15 core active.	Package
63:56	Maximum Ratio Limit for16C Maximum turbo ratio limit of 16 core active.	Package
Register Address: 1AFH, 431	MSR_TURBO_RATIO_LIMIT2	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 17C Maximum turbo ratio limit of 17 core active.	Package
15:8	Maximum Ratio Limit for 18C Maximum turbo ratio limit of 18 core active.	Package
62:16	Reserved.	Package
63	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1, and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).	Package
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from the Intel QPI 0 module.		Package
Register Address: 415H, 1045	IA32_MC5_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from the Intel QPI 0 module.		Package

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI 0 module.		Package
Register Address: 417H, 1047	IA32_MC5_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI 0 module.		Package
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41BH, 1051	IA32_MC6_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41CH, 1052	IA32_MC7_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41DH, 1053	IA32_MC7_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41EH, 1054	IA32_MC7_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41FH, 1055	IA32_MC7_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 420H, 1056	IA32_MC8_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC8 reports MC errors from the home agent HA 1.		Package
Register Address: 421H, 1057	IA32_MC8_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC8 reports MC errors from the home agent HA 1.		Package
Register Address: 422H, 1058	IA32_MC8_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC8 reports MC errors from the home agent HA 1.		Package

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 423H, 1059	IA32_MC8_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the home agent HA 1.		Package
Register Address: 424H, 1060	IA32_MC9_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 425H, 1061	IA32_MC9_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 426H, 1062	IA32_MC9_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 427H, 1063	IA32_MC9_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 428H, 1064	IA32_MC10_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 429H, 1065	IA32_MC10_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42AH, 1066	IA32_MC10_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42BH, 1067	IA32_MC10_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42CH, 1068	IA32_MC11_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42DH, 1069	IA32_MC11_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42EH, 1070	IA32_MC11_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42FH, 1071	IA32_MC11_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 430H, 1072	IA32_MC12_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 431H, 1073	IA32_MC12_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 432H, 1074	IA32_MC12_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 433H, 1075	IA32_MC12_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 434H, 1076	IA32_MC13_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 435H, 1077	IA32_MC13_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 436H, 1078	IA32_MC13_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 437H, 1079	IA32_MC13_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 438H, 1080	IA32_MC14_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 439H, 1081	IA32_MC14_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43AH, 1082	IA32_MC14_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43BH, 1083	IA32_MC14_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43CH, 1084	IA32_MC15_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 43DH, 1085	IA32_MC15_STATUS	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.	Package
Register Address: 43EH, 1086	IA32_MC15_ADDR	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.	Package
Register Address: 43FH, 1087	IA32_MC15_MISC	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.	Package
Register Address: 440H, 1088	IA32_MC16_CTL	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.	Package
Register Address: 441H, 1089	IA32_MC16_STATUS	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.	Package
Register Address: 442H, 1090	IA32_MC16_ADDR	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.	Package
Register Address: 443H, 1091	IA32_MC16_MISC	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.	Package
Register Address: 444H, 1092	IA32_MC17_CTL	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.	Package
Register Address: 445H, 1093	IA32_MC17_STATUS	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.	Package
Register Address: 446H, 1094	IA32_MC17_ADDR	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.	Package
Register Address: 447H, 1095	IA32_MC17_MISC	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.	Package
Register Address: 448H, 1096	IA32_MC18_CTL	
	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.	Package

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 449H, 1097	IA32_MC18_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44AH, 1098	IA32_MC18_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44BH, 1099	IA32_MC18_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44CH, 1100	IA32_MC19_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44DH, 1101	IA32_MC19_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44EH, 1102	IA32_MC19_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44FH, 1103	IA32_MC19_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 450H, 1104	IA32_MC20_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 451H, 1105	IA32_MC20_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 452H, 1106	IA32_MC20_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 453H, 1107	IA32_MC20_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 454H, 1108	IA32_MC21_CTL	

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: 455H, 1109	IA32_MC21_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: 456H, 1110	IA32_MC21_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: 457H, 1111	IA32_MC21_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers Used in RAPL Interfaces (R/O)		Package
3:0	Power Units See Section 16.10.1, "RAPL Interfaces."	Package
7:4	Reserved.	Package
12:8	Energy Status Units Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).	Package
15:13	Reserved.	Package
19:16	Time Units See Section 16.10.1, "RAPL Interfaces."	Package
63:20	Reserved.	
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
DRAM RAPL Power Limit Control (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) Energy Consumed by DRAM devices.		Package
31:0	Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).	
63:32	Reserved.	
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	
DRAM RAPL Parameters (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61EH, 1566	MSR_PCIE_PLL_RATIO	

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Configuration of PCIE PLL Relative to BCLK(R/W)		Package
1:0	PCIE Ratio (R/W) 00b: Use 5:5 mapping for100MHz operation (default). 01b: Use 5:4 mapping for125MHz operation. 10b: Use 5:3 mapping for166MHz operation. 11b: Use 5:2 mapping for250MHz operation.	Package
2	LPLL Select (R/W) If 1, use configured setting of PCIE Ratio.	Package
3	LONG RESET (R/W) If 1, wait an additional time-out before re-locking Gen2/Gen3 PLLs.	Package
63:4	Reserved.	
Register Address: 620H, 1568	MSR_UNCORE_RATIO_LIMIT	
Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.		Package
6:0	MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.	
7	Reserved.	
14:8	MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.	
63:15	Reserved.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
Reserved (R/O) Reads return 0.		Package
Register Address: 690H, 1680	MSR_CORE_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)		Package
0	PROCHOT Status (R0) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.	
2	Power Budget Management Status (R0) When set, frequency is reduced below the operating system request due to PBM limit	
3	Platform Configuration Services Status (R0) When set, frequency is reduced below the operating system request due to PCS limit	
4	Reserved.	

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	Bit Description	Scope
5	Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.	
6	VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.	
7	Reserved.	
8	Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).	
9	Reserved.	
10	Multi-Core Turbo Status (R0) When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits.	
12:11	Reserved.	
13	Core Frequency P1 Status (R0) When set, frequency is reduced below max non-turbo P1.	
14	Core Max N-Core Turbo Frequency Limiting Status (R0) When set, frequency is reduced below max n-core turbo frequency.	
15	Core Frequency Limiting Status (R0) When set, frequency is reduced below the operating system request.	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
18	Power Budget Management Log When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19	Platform Configuration Services Log When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
20	Reserved.	
21	Autonomous Utilization-Based Frequency Control Log When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	Reserved.	
24	Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Reserved.	
26	Multi-Core Turbo Log When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28:27	Reserved.	
29	Core Frequency P1 Log When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
30	Core Max N-Core Turbo Frequency Limiting Log When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
31	Core Frequency Limiting Log When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:32	Reserved.	
Register Address: C8DH, 3213	IA32_QM_EVTSEL	
Monitoring Event Select Register (R/W) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.		Thread
7:0	EventID (R/W) Event encoding: 0x0: No monitoring. 0x1: L3 occupancy monitoring. All other encoding reserved.	
31:8	Reserved.	
41:32	RMID (R/W)	
63:42	Reserved.	
Register Address: C8EH, 3214	IA32_QM_CTR	
Monitoring Counter Register (R/O) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.		Thread

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
61:0	Resource Monitored Data	
62	Unavailable: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.	
63	Error: If 1, indicates an unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	
Register Address: C8FH, 3215	IA32_PQR_ASSOC	
Resource Association Register (R/W)		Thread
9:0	RMID	
63: 10	Reserved.	
See Table 2-20 and Table 2-29 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3FH.		

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.14.1 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family

The Intel Xeon Processor E5 v3 and E7 v3 families are based on Haswell-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-33. For complete details of the uncore PMU, refer to the Intel Xeon Processor E5 v3 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3FH.

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 700H, 1792	MSR_PMON_GLOBAL_CTL	
Uncore PerfMon Per-Socket Global Control		Package
Register Address: 701H, 1793	MSR_PMON_GLOBAL_STATUS	
Uncore PerfMon Per-Socket Global Status		Package
Register Address: 702H, 1794	MSR_PMON_GLOBAL_CONFIG	
Uncore PerfMon Per-Socket Global Configuration		Package
Register Address: 703H, 1795	MSR_U_PMON_UCLK_FIXED_CTL	
Uncore U-Box UCLK Fixed Counter Control		Package
Register Address: 704H, 1796	MSR_U_PMON_UCLK_FIXED_CTR	
Uncore U-Box UCLK Fixed Counter		Package
Register Address: 705H, 1797	MSR_U_PMON_EVNTSEL0	
Uncore U-Box PerfMon Event Select for U-Box Counter 0		Package
Register Address: 706H, 1798	MSR_U_PMON_EVNTSEL1	
Uncore U-Box PerfMon Event Select for U-Box Counter 1		Package
Register Address: 708H, 1800	MSR_U_PMON_BOX_STATUS	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore U-Box PerfMon U-Box Wide Status		Package
Register Address: 709H, 1801	MSR_U_PMON_CTR0	
Uncore U-Box PerfMon Counter 0		Package
Register Address: 70AH, 1802	MSR_U_PMON_CTR1	
Uncore U-Box PerfMon Counter 1		Package
Register Address: 710H, 1808	MSR_PCU_PMON_BOX_CTL	
Uncore PCU PerfMon for PCU-Box-Wide Control		Package
Register Address: 711H, 1809	MSR_PCU_PMON_EVNTSEL0	
Uncore PCU PerfMon Event Select for PCU Counter 0		Package
Register Address: 712H, 1810	MSR_PCU_PMON_EVNTSEL1	
Uncore PCU PerfMon Event Select for PCU Counter 1		Package
Register Address: 713H, 1811	MSR_PCU_PMON_EVNTSEL2	
Uncore PCU PerfMon Event Select for PCU Counter 2		Package
Register Address: 714H, 1812	MSR_PCU_PMON_EVNTSEL3	
Uncore PCU PerfMon Event Select for PCU Counter 3		Package
Register Address: 715H, 1813	MSR_PCU_PMON_BOX_FILTER	
Uncore PCU PerfMon Box-Wide Filter		Package
Register Address: 716H, 1814	MSR_PCU_PMON_BOX_STATUS	
Uncore PCU PerfMon Box Wide Status		Package
Register Address: 717H, 1815	MSR_PCU_PMON_CTR0	
Uncore PCU PerfMon Counter 0		Package
Register Address: 718H, 1816	MSR_PCU_PMON_CTR1	
Uncore PCU PerfMon Counter 1		Package
Register Address: 719H, 1817	MSR_PCU_PMON_CTR2	
Uncore PCU PerfMon Counter 2		Package
Register Address: 71AH, 1818	MSR_PCU_PMON_CTR3	
Uncore PCU PerfMon Counter 3		Package
Register Address: 720H, 1824	MSR_S0_PMON_BOX_CTL	
Uncore SBo 0 PerfMon for SBo 0 Box-Wide Control		Package
Register Address: 721H, 1825	MSR_S0_PMON_EVNTSEL0	
Uncore SBo 0 PerfMon Event Select for SBo 0 Counter 0		Package
Register Address: 722H, 1826	MSR_S0_PMON_EVNTSEL1	
Uncore SBo 0 PerfMon Event Select for SBo 0 Counter 1		Package
Register Address: 723H, 1827	MSR_S0_PMON_EVNTSEL2	
Uncore SBo 0 PerfMon Event Select for SBo 0 Counter 2		Package
Register Address: 724H, 1828	MSR_S0_PMON_EVNTSEL3	
Uncore SBo 0 PerfMon Event Select for SBo 0 Counter 3		Package

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 725H, 1829	MSR_S0_PMON_BOX_FILTER	
Uncore SBo 0 PerfMon Box-Wide Filter		Package
Register Address: 726H, 1830	MSR_S0_PMON_CTR0	
Uncore SBo 0 PerfMon Counter 0		Package
Register Address: 727H, 1831	MSR_S0_PMON_CTR1	
Uncore SBo 0 PerfMon Counter 1		Package
Register Address: 728H, 1832	MSR_S0_PMON_CTR2	
Uncore SBo 0 PerfMon Counter 2		Package
Register Address: 729H, 1833	MSR_S0_PMON_CTR3	
Uncore SBo 0 PerfMon Counter 3		Package
Register Address: 72AH, 1834	MSR_S1_PMON_BOX_CTL	
Uncore SBo 1 PerfMon for SBo 1 Box-Wide Control		Package
Register Address: 72BH, 1835	MSR_S1_PMON_EVNTSEL0	
Uncore SBo 1 PerfMon Event Select for SBo 1 Counter 0		Package
Register Address: 72CH, 1836	MSR_S1_PMON_EVNTSEL1	
Uncore SBo 1 PerfMon Event Select for SBo 1 Counter 1		Package
Register Address: 72DH, 1837	MSR_S1_PMON_EVNTSEL2	
Uncore SBo 1 PerfMon Event Select for SBo 1 Counter 2		Package
Register Address: 72EH, 1838	MSR_S1_PMON_EVNTSEL3	
Uncore SBo 1 PerfMon Event Select for SBo 1 Counter 3		Package
Register Address: 72FH, 1839	MSR_S1_PMON_BOX_FILTER	
Uncore SBo 1 PerfMon Box-Wide Filter		Package
Register Address: 730H, 1840	MSR_S1_PMON_CTR0	
Uncore SBo 1 PerfMon Counter 0		Package
Register Address: 731H, 1841	MSR_S1_PMON_CTR1	
Uncore SBo 1 PerfMon Counter 1		Package
Register Address: 732H, 1842	MSR_S1_PMON_CTR2	
Uncore SBo 1 PerfMon Counter 2		Package
Register Address: 733H, 1843	MSR_S1_PMON_CTR3	
Uncore SBo 1 PerfMon Counter 3		Package
Register Address: 734H, 1844	MSR_S2_PMON_BOX_CTL	
Uncore SBo 2 PerfMon for SBo 2 Box-Wide Control		Package
Register Address: 735H, 1845	MSR_S2_PMON_EVNTSEL0	
Uncore SBo 2 PerfMon Event Select for SBo 2 Counter 0		Package
Register Address: 736H, 1846	MSR_S2_PMON_EVNTSEL1	
Uncore SBo 2 PerfMon Event Select for SBo 2 Counter 1		Package
Register Address: 737H, 1847	MSR_S2_PMON_EVNTSEL2	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore SBo 2 PerfMon Event Select for SBo 2 Counter 2		Package
Register Address: 738H, 1848	MSR_S2_PMON_EVNTSEL3	
Uncore SBo 2 PerfMon Event Select for SBo 2 Counter 3		Package
Register Address: 739H, 1849	MSR_S2_PMON_BOX_FILTER	
Uncore SBo 2 PerfMon Box-Wide Filter		Package
Register Address: 73AH, 1850	MSR_S2_PMON_CTR0	
Uncore SBo 2 PerfMon Counter 0		Package
Register Address: 73BH, 1851	MSR_S2_PMON_CTR1	
Uncore SBo 2 PerfMon Counter 1		Package
Register Address: 73CH, 1852	MSR_S2_PMON_CTR2	
Uncore SBo 2 PerfMon Counter 2		Package
Register Address: 73DH, 1853	MSR_S2_PMON_CTR3	
Uncore SBo 2 PerfMon Counter 3		Package
Register Address: 73EH, 1854	MSR_S3_PMON_BOX_CTL	
Uncore SBo 3 PerfMon for SBo 3 Box-Wide Control		Package
Register Address: 73FH, 1855	MSR_S3_PMON_EVNTSEL0	
Uncore SBo 3 PerfMon Event Select for SBo 3 Counter 0		Package
Register Address: 740H, 1856	MSR_S3_PMON_EVNTSEL1	
Uncore SBo 3 PerfMon Event Select for SBo 3 Counter 1		Package
Register Address: 741H, 1857	MSR_S3_PMON_EVNTSEL2	
Uncore SBo 3 PerfMon Event Select for SBo 3 Counter 2		Package
Register Address: 742H, 1858	MSR_S3_PMON_EVNTSEL3	
Uncore SBo 3 PerfMon Event Select for SBo 3 Counter 3		Package
Register Address: 743H, 1859	MSR_S3_PMON_BOX_FILTER	
Uncore SBo 3 PerfMon Box-Wide Filter		Package
Register Address: 744H, 1860	MSR_S3_PMON_CTR0	
Uncore SBo 3 PerfMon Counter 0		Package
Register Address: 745H, 1861	MSR_S3_PMON_CTR1	
Uncore SBo 3 PerfMon Counter 1		Package
Register Address: 746H, 1862	MSR_S3_PMON_CTR2	
Uncore SBo 3 PerfMon Counter 2		Package
Register Address: 747H, 1863	MSR_S3_PMON_CTR3	
Uncore SBo 3 PerfMon Counter 3		Package
Register Address: E00H, 3584	MSR_C0_PMON_BOX_CTL	
Uncore C-Box 0 PerfMon for Box-Wide Control		Package
Register Address: E01H, 3585	MSR_C0_PMON_EVNTSEL0	
Uncore C-Box 0 PerfMon Event Select for C-Box 0 Counter 0		Package

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E02H, 3586	MSR_C0_PMON_EVNTSEL1	
Uncore C-Box 0 PerfMon Event Select for C-Box 0 Counter 1		Package
Register Address: E03H, 3587	MSR_C0_PMON_EVNTSEL2	
Uncore C-Box 0 PerfMon Event Select for C-Box 0 Counter 2		Package
Register Address: E04H, 3588	MSR_C0_PMON_EVNTSEL3	
Uncore C-Box 0 PerfMon Event Select for C-Box 0 Counter 3		Package
Register Address: E05H, 3589	MSR_C0_PMON_BOX_FILTER0	
Uncore C-Box 0 PerfMon Box Wide Filter 0		Package
Register Address: E06H, 3590	MSR_C0_PMON_BOX_FILTER1	
Uncore C-Box 0 PerfMon Box Wide Filter 1		Package
Register Address: E07H, 3591	MSR_C0_PMON_BOX_STATUS	
Uncore C-Box 0 PerfMon Box Wide Status		Package
Register Address: E08H, 3592	MSR_C0_PMON_CTR0	
Uncore C-Box 0 PerfMon Counter 0		Package
Register Address: E09H, 3593	MSR_C0_PMON_CTR1	
Uncore C-Box 0 PerfMon Counter 1		Package
Register Address: E0AH, 3594	MSR_C0_PMON_CTR2	
Uncore C-Box 0 PerfMon Counter 2		Package
Register Address: E0BH, 3595	MSR_C0_PMON_CTR3	
Uncore C-Box 0 PerfMon Counter 3		Package
Register Address: E10H, 3600	MSR_C1_PMON_BOX_CTL	
Uncore C-Box 1 PerfMon for Box-Wide Control		Package
Register Address: E11H, 3601	MSR_C1_PMON_EVNTSEL0	
Uncore C-Box 1 PerfMon Event Select for C-Box 1 Counter 0		Package
Register Address: E12H, 3602	MSR_C1_PMON_EVNTSEL1	
Uncore C-Box 1 PerfMon Event Select for C-Box 1 Counter 1		Package
Register Address: E13H, 3603	MSR_C1_PMON_EVNTSEL2	
Uncore C-Box 1 PerfMon Event Select for C-Box 1 Counter 2		Package
Register Address: E14H, 3604	MSR_C1_PMON_EVNTSEL3	
Uncore C-Box 1 PerfMon Event Select for C-Box 1 Counter 3		Package
Register Address: E15H, 3605	MSR_C1_PMON_BOX_FILTER0	
Uncore C-Box 1 PerfMon Box Wide Filter 0		Package
Register Address: E16H, 3606	MSR_C1_PMON_BOX_FILTER1	
Uncore C-Box 1 PerfMon Box Wide Filter1		Package
Register Address: E17H, 3607	MSR_C1_PMON_BOX_STATUS	
Uncore C-Box 1 PerfMon Box Wide Status		Package
Register Address: E18H, 3608	MSR_C1_PMON_CTR0	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 1 PerfMon Counter 0		Package
Register Address: E19H, 3609	MSR_C1_PMON_CTR1	
Uncore C-Box 1 PerfMon Counter 1		Package
Register Address: E1AH, 3610	MSR_C1_PMON_CTR2	
Uncore C-Box 1 PerfMon Counter 2		Package
Register Address: E1BH, 3611	MSR_C1_PMON_CTR3	
Uncore C-Box 1 PerfMon Counter 3		Package
Register Address: E20H, 3616	MSR_C2_PMON_BOX_CTL	
Uncore C-Box 2 PerfMon for Box-Wide Control		Package
Register Address: E21H, 3617	MSR_C2_PMON_EVNTSEL0	
Uncore C-Box 2 PerfMon Event Select for C-Box 2 Counter 0		Package
Register Address: E22H, 3618	MSR_C2_PMON_EVNTSEL1	
Uncore C-Box 2 PerfMon Event Select for C-Box 2 Counter 1		Package
Register Address: E23H, 3619	MSR_C2_PMON_EVNTSEL2	
Uncore C-Box 2 PerfMon Event Select for C-Box 2 Counter 2		Package
Register Address: E24H, 3620	MSR_C2_PMON_EVNTSEL3	
Uncore C-Box 2 PerfMon Event select for C-Box 2 Counter 3		Package
Register Address: E25H, 3621	MSR_C2_PMON_BOX_FILTER0	
Uncore C-Box 2 PerfMon Box Wide Filter 0		Package
Register Address: E26H, 3622	MSR_C2_PMON_BOX_FILTER1	
Uncore C-Box 2 PerfMon Box Wide Filter1		Package
Register Address: E27H, 3623	MSR_C2_PMON_BOX_STATUS	
Uncore C-Box 2 PerfMon Box Wide Status		Package
Register Address: E28H, 3624	MSR_C2_PMON_CTR0	
Uncore C-Box 2 PerfMon Counter 0		Package
Register Address: E29H, 3625	MSR_C2_PMON_CTR1	
Uncore C-Box 2 PerfMon Counter 1		Package
Register Address: E2AH, 3626	MSR_C2_PMON_CTR2	
Uncore C-Box 2 PerfMon Counter 2		Package
Register Address: E2BH, 3627	MSR_C2_PMON_CTR3	
Uncore C-Box 2 PerfMon Counter 3		Package
Register Address: E30H, 3632	MSR_C3_PMON_BOX_CTL	
Uncore C-Box 3 PerfMon for Box-Wide Control		Package
Register Address: E31H, 3633	MSR_C3_PMON_EVNTSEL0	
Uncore C-Box 3 PerfMon Event Select for C-Box 3 Counter 0		Package
Register Address: E32H, 3634	MSR_C3_PMON_EVNTSEL1	
Uncore C-Box 3 PerfMon Event Select for C-Box 3 Counter 1		Package

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E33H, 3635	MSR_C3_PMON_EVNTSEL2	
Uncore C-Box 3 PerfMon Event Select for C-Box 3 Counter 2		Package
Register Address: E34H, 3636	MSR_C3_PMON_EVNTSEL3	
Uncore C-Box 3 PerfMon Event Select for C-Box 3 Counter 3		Package
Register Address: E35H, 3637	MSR_C3_PMON_BOX_FILTER0	
Uncore C-Box 3 PerfMon Box Wide Filter 0		Package
Register Address: E36H, 3638	MSR_C3_PMON_BOX_FILTER1	
Uncore C-Box 3 PerfMon Box Wide Filter1		Package
Register Address: E37H, 3639	MSR_C3_PMON_BOX_STATUS	
Uncore C-Box 3 PerfMon Box Wide Status		Package
Register Address: E38H, 3640	MSR_C3_PMON_CTR0	
Uncore C-Box 3 PerfMon Counter 0		Package
Register Address: E39H, 3641	MSR_C3_PMON_CTR1	
Uncore C-Box 3 PerfMon Counter 1		Package
Register Address: E3AH, 3642	MSR_C3_PMON_CTR2	
Uncore C-Box 3 PerfMon Counter 2		Package
Register Address: E3BH, 3643	MSR_C3_PMON_CTR3	
Uncore C-Box 3 PerfMon Counter 3		Package
Register Address: E40H, 3648	MSR_C4_PMON_BOX_CTL	
Uncore C-Box 4 PerfMon for Box-Wide Control		Package
Register Address: E41H, 3649	MSR_C4_PMON_EVNTSEL0	
Uncore C-Box 4 PerfMon Event Select for C-Box 4 Counter 0		Package
Register Address: E42H, 3650	MSR_C4_PMON_EVNTSEL1	
Uncore C-Box 4 PerfMon Event Select for C-Box 4 Counter 1		Package
Register Address: E43H, 3651	MSR_C4_PMON_EVNTSEL2	
Uncore C-Box 4 PerfMon Event Select for C-Box 4 Counter 2		Package
Register Address: E44H, 3652	MSR_C4_PMON_EVNTSEL3	
Uncore C-Box 4 PerfMon Event Select for C-Box 4 Counter 3		Package
Register Address: E45H, 3653	MSR_C4_PMON_BOX_FILTER0	
Uncore C-Box 4 PerfMon Box Wide Filter 0		Package
Register Address: E46H, 3654	MSR_C4_PMON_BOX_FILTER1	
Uncore C-Box 4 PerfMon Box Wide Filter1		Package
Register Address: E47H, 3655	MSR_C4_PMON_BOX_STATUS	
Uncore C-Box 4 PerfMon Box Wide Status		Package
Register Address: E48H, 3656	MSR_C4_PMON_CTR0	
Uncore C-Box 4 PerfMon Counter 0		Package
Register Address: E49H, 3657	MSR_C4_PMON_CTR1	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 4 PerfMon Counter 1		Package
Register Address: E4AH, 3658	MSR_C4_PMON_CTR2	
Uncore C-Box 4 PerfMon Counter 2		Package
Register Address: E4BH, 3659	MSR_C4_PMON_CTR3	
Uncore C-Box 4 PerfMon Counter 3		Package
Register Address: E50H, 3664	MSR_C5_PMON_BOX_CTL	
Uncore C-Box 5 PerfMon for Box-Wide Control		Package
Register Address: E51H, 3665	MSR_C5_PMON_EVNTSEL0	
Uncore C-Box 5 PerfMon Event Select for C-Box 5 Counter 0		Package
Register Address: E52H, 3666	MSR_C5_PMON_EVNTSEL1	
Uncore C-Box 5 PerfMon Event Select for C-Box 5 Counter 1		Package
Register Address: E53H, 3667	MSR_C5_PMON_EVNTSEL2	
Uncore C-Box 5 PerfMon Event Select for C-Box 5 Counter 2		Package
Register Address: E54H, 3668	MSR_C5_PMON_EVNTSEL3	
Uncore C-Box 5 PerfMon Event Select for C-Box 5 Counter 3		Package
Register Address: E55H, 3669	MSR_C5_PMON_BOX_FILTER0	
Uncore C-Box 5 PerfMon Box Wide Filter 0		Package
Register Address: E56H, 3670	MSR_C5_PMON_BOX_FILTER1	
Uncore C-Box 5 PerfMon Box Wide Filter 1		Package
Register Address: E57H, 3671	MSR_C5_PMON_BOX_STATUS	
Uncore C-Box 5 PerfMon Box Wide Status		Package
Register Address: E58H, 3672	MSR_C5_PMON_CTR0	
Uncore C-Box 5 PerfMon Counter 0		Package
Register Address: E59H, 3673	MSR_C5_PMON_CTR1	
Uncore C-Box 5 PerfMon Counter 1		Package
Register Address: E5AH, 3674	MSR_C5_PMON_CTR2	
Uncore C-Box 5 PerfMon Counter 2		Package
Register Address: E5BH, 3675	MSR_C5_PMON_CTR3	
Uncore C-Box 5 PerfMon Counter 3		Package
Register Address: E60H, 3680	MSR_C6_PMON_BOX_CTL	
Uncore C-Box 6 PerfMon for Box-Wide Control		Package
Register Address: E61H, 3681	MSR_C6_PMON_EVNTSEL0	
Uncore C-Box 6 PerfMon Event Select for C-Box 6 Counter 0		Package
Register Address: E62H, 3682	MSR_C6_PMON_EVNTSEL1	
Uncore C-Box 6 PerfMon Event Select for C-Box 6 Counter 1		Package
Register Address: E63H, 3683	MSR_C6_PMON_EVNTSEL2	
Uncore C-Box 6 PerfMon Event Select for C-Box 6 Counter 2		Package

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E64H, 3684	MSR_C6_PMON_EVNTSEL3	
Uncore C-Box 6 PerfMon Event Select for C-Box 6 Counter 3		Package
Register Address: E65H, 3685	MSR_C6_PMON_BOX_FILTER0	
Uncore C-Box 6 PerfMon Box Wide Filter 0		Package
Register Address: E66H, 3686	MSR_C6_PMON_BOX_FILTER1	
Uncore C-Box 6 PerfMon Box Wide Filter 1		Package
Register Address: E67H, 3687	MSR_C6_PMON_BOX_STATUS	
Uncore C-Box 6 PerfMon Box Wide Status		Package
Register Address: E68H, 3688	MSR_C6_PMON_CTR0	
Uncore C-Box 6 PerfMon Counter 0		Package
Register Address: E69H, 3689	MSR_C6_PMON_CTR1	
Uncore C-Box 6 PerfMon Counter 1		Package
Register Address: E6AH, 3690	MSR_C6_PMON_CTR2	
Uncore C-Box 6 PerfMon Counter 2		Package
Register Address: E6BH, 3691	MSR_C6_PMON_CTR3	
Uncore C-Box 6 PerfMon Counter 3		Package
Register Address: E70H, 3696	MSR_C7_PMON_BOX_CTL	
Uncore C-Box 7 PerfMon for Box-Wide Control		Package
Register Address: E71H, 3697	MSR_C7_PMON_EVNTSEL0	
Uncore C-Box 7 PerfMon Event Select for C-Box 7 Counter 0		Package
Register Address: E72H, 3698	MSR_C7_PMON_EVNTSEL1	
Uncore C-Box 7 PerfMon Event Select for C-Box 7 Counter 1		Package
Register Address: E73H, 3699	MSR_C7_PMON_EVNTSEL2	
Uncore C-Box 7 PerfMon Event Select for C-Box 7 Counter 2		Package
Register Address: E74H, 3700	MSR_C7_PMON_EVNTSEL3	
Uncore C-Box 7 PerfMon Event Select for C-Box 7 Counter 3		Package
Register Address: E75H, 3701	MSR_C7_PMON_BOX_FILTER0	
Uncore C-Box 7 PerfMon Box Wide Filter 0		Package
Register Address: E76H, 3702	MSR_C7_PMON_BOX_FILTER1	
Uncore C-Box 7 PerfMon Box Wide Filter 1		Package
Register Address: E77H, 3703	MSR_C7_PMON_BOX_STATUS	
Uncore C-Box 7 PerfMon Box Wide Status		Package
Register Address: E78H, 3704	MSR_C7_PMON_CTR0	
Uncore C-Box 7 PerfMon Counter 0		Package
Register Address: E79H, 3705	MSR_C7_PMON_CTR1	
Uncore C-Box 7 PerfMon Counter 1		Package
Register Address: E7AH, 3706	MSR_C7_PMON_CTR2	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 7 PerfMon Counter 2		Package
Register Address: E7BH, 3707	MSR_C7_PMON_CTR3	
Uncore C-Box 7 PerfMon Counter 3		Package
Register Address: E80H, 3712	MSR_C8_PMON_BOX_CTL	
Uncore C-Box 8 PerfMon Local Box Wide Control		Package
Register Address: E81H, 3713	MSR_C8_PMON_EVNTSEL0	
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 0		Package
Register Address: E82H, 3714	MSR_C8_PMON_EVNTSEL1	
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 1		Package
Register Address: E83H, 3715	MSR_C8_PMON_EVNTSEL2	
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 2		Package
Register Address: E84H, 3716	MSR_C8_PMON_EVNTSEL3	
Uncore C-Box 8 PerfMon Event Select for C-Box 8 Counter 3		Package
Register Address: E85H, 3717	MSR_C8_PMON_BOX_FILTER0	
Uncore C-Box 8 PerfMon Box Wide Filter 0		Package
Register Address: E86H, 3718	MSR_C8_PMON_BOX_FILTER1	
Uncore C-Box 8 PerfMon Box Wide Filter 1		Package
Register Address: E87H, 3719	MSR_C8_PMON_BOX_STATUS	
Uncore C-Box 8 PerfMon Box Wide Status		Package
Register Address: E88H, 3720	MSR_C8_PMON_CTR0	
Uncore C-Box 8 PerfMon Counter 0		Package
Register Address: E89H, 3721	MSR_C8_PMON_CTR1	
Uncore C-Box 8 PerfMon Counter 1		Package
Register Address: E8AH, 3722	MSR_C8_PMON_CTR2	
Uncore C-Box 8 PerfMon Counter 2		Package
Register Address: E8BH, 3723	MSR_C8_PMON_CTR3	
Uncore C-Box 8 PerfMon Counter 3		Package
Register Address: E90H, 3728	MSR_C9_PMON_BOX_CTL	
Uncore C-Box 9 PerfMon Local Box Wide Control		Package
Register Address: E91H, 3729	MSR_C9_PMON_EVNTSEL0	
Uncore C-Box 9 PerfMon Event Select for C-Box 9 Counter 0		Package
Register Address: E92H, 3730	MSR_C9_PMON_EVNTSEL1	
Uncore C-Box 9 PerfMon Event Select for C-Box 9 Counter 1		Package
Register Address: E93H, 3731	MSR_C9_PMON_EVNTSEL2	
Uncore C-Box 9 PerfMon Event Select for C-Box 9 Counter 2		Package
Register Address: E94H, 3732	MSR_C9_PMON_EVNTSEL3	
Uncore C-Box 9 PerfMon Event Select for C-Box 9 Counter 3		Package

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E95H, 3733	MSR_C9_PMON_BOX_FILTER0	
Uncore C-Box 9 PerfMon Box Wide Filter 0		Package
Register Address: E96H, 3734	MSR_C9_PMON_BOX_FILTER1	
Uncore C-Box 9 PerfMon Box Wide Filter 1		Package
Register Address: E97H, 3735	MSR_C9_PMON_BOX_STATUS	
Uncore C-Box 9 PerfMon Box Wide Status		Package
Register Address: E98H, 3736	MSR_C9_PMON_CTR0	
Uncore C-Box 9 PerfMon Counter 0		Package
Register Address: E99H, 3737	MSR_C9_PMON_CTR1	
Uncore C-Box 9 PerfMon Counter 1		Package
Register Address: E9AH, 3738	MSR_C9_PMON_CTR2	
Uncore C-Box 9 PerfMon Counter 2		Package
Register Address: E9BH, 3739	MSR_C9_PMON_CTR3	
Uncore C-Box 9 PerfMon Counter 3		Package
Register Address: EA0H, 3744	MSR_C10_PMON_BOX_CTL	
Uncore C-Box 10 PerfMon Local Box Wide Control		Package
Register Address: EA1H, 3745	MSR_C10_PMON_EVNTSEL0	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 0		Package
Register Address: EA2H, 3746	MSR_C10_PMON_EVNTSEL1	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 1		Package
Register Address: EA3H, 3747	MSR_C10_PMON_EVNTSEL2	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 2		Package
Register Address: EA4H, 3748	MSR_C10_PMON_EVNTSEL3	
Uncore C-Box 10 PerfMon Event Select for C-Box 10 Counter 3		Package
Register Address: EA5H, 3749	MSR_C10_PMON_BOX_FILTER0	
Uncore C-Box 10 PerfMon Box Wide Filter 0		Package
Register Address: EA6H, 3750	MSR_C10_PMON_BOX_FILTER1	
Uncore C-Box 10 PerfMon Box Wide Filter 1		Package
Register Address: EA7H, 3751	MSR_C10_PMON_BOX_STATUS	
Uncore C-Box 10 PerfMon Box Wide Status		Package
Register Address: EA8H, 3752	MSR_C10_PMON_CTR0	
Uncore C-Box 10 PerfMon Counter 0		Package
Register Address: EA9H, 3753	MSR_C10_PMON_CTR1	
Uncore C-Box 10 PerfMon Counter 1		Package
Register Address: EAAH, 3754	MSR_C10_PMON_CTR2	
Uncore C-Box 10 PerfMon Counter 2		Package
Register Address: EABH, 3755	MSR_C10_PMON_CTR3	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 10 PerfMon Counter 3		Package
Register Address: EB0H, 3760	MSR_C11_PMON_BOX_CTL	
Uncore C-Box 11 PerfMon Local Box Wide Control		Package
Register Address: EB1H, 3761	MSR_C11_PMON_EVNTSEL0	
Uncore C-Box 11 PerfMon Event Select for C-Box 11 Counter 0		Package
Register Address: EB2H, 3762	MSR_C11_PMON_EVNTSEL1	
Uncore C-Box 11 PerfMon Event Select for C-Box 11 Counter 1		Package
Register Address: EB3H, 3763	MSR_C11_PMON_EVNTSEL2	
Uncore C-Box 11 PerfMon Event Select for C-Box 11 Counter 2		Package
Register Address: EB4H, 3764	MSR_C11_PMON_EVNTSEL3	
Uncore C-box 11 PerfMon Event Select for C-Box 11 Counter 3		Package
Register Address: EB5H, 3765	MSR_C11_PMON_BOX_FILTER0	
Uncore C-Box 11 PerfMon Box Wide Filter 0		Package
Register Address: EB6H, 3766	MSR_C11_PMON_BOX_FILTER1	
Uncore C-Box 11 PerfMon Box Wide Filter 1		Package
Register Address: EB7H, 3767	MSR_C11_PMON_BOX_STATUS	
Uncore C-Box 11 PerfMon Box Wide Status		Package
Register Address: EB8H, 3768	MSR_C11_PMON_CTR0	
Uncore C-Box 11 PerfMon Counter 0		Package
Register Address: EB9H, 3769	MSR_C11_PMON_CTR1	
Uncore C-Box 11 PerfMon Counter 1		Package
Register Address: EBAH, 3770	MSR_C11_PMON_CTR2	
Uncore C-Box 11 PerfMon Counter 2		Package
Register Address: EBBH, 3771	MSR_C11_PMON_CTR3	
Uncore C-Box 11 PerfMon Counter 3		Package
Register Address: EC0H, 3776	MSR_C12_PMON_BOX_CTL	
Uncore C-Box 12 PerfMon Local Box Wide Control		Package
Register Address: EC1H, 3777	MSR_C12_PMON_EVNTSEL0	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 0		Package
Register Address: EC2H, 3778	MSR_C12_PMON_EVNTSEL1	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 1		Package
Register Address: EC3H, 3779	MSR_C12_PMON_EVNTSEL2	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 2		Package
Register Address: EC4H, 3780	MSR_C12_PMON_EVNTSEL3	
Uncore C-Box 12 PerfMon Event Select for C-Box 12 Counter 3		Package
Register Address: EC5H, 3781	MSR_C12_PMON_BOX_FILTER0	
Uncore C-Box 12 PerfMon Box Wide Filter 0		Package

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: EC6H, 3782	MSR_C12_PMON_BOX_FILTER1	
Uncore C-Box 12 PerfMon Box Wide Filter 1		Package
Register Address: EC7H, 3783	MSR_C12_PMON_BOX_STATUS	
Uncore C-Box 12 PerfMon Box Wide Status		Package
Register Address: EC8H, 3784	MSR_C12_PMON_CTR0	
Uncore C-Box 12 PerfMon Counter 0		Package
Register Address: EC9H, 3785	MSR_C12_PMON_CTR1	
Uncore C-Box 12 PerfMon Counter 1		Package
Register Address: ECAH, 3786	MSR_C12_PMON_CTR2	
Uncore C-Box 12 PerfMon Counter 2		Package
Register Address: ECBH, 3787	MSR_C12_PMON_CTR3	
Uncore C-Box 12 PerfMon Counter 3		Package
Register Address: ED0H, 3792	MSR_C13_PMON_BOX_CTL	
Uncore C-Box 13 PerfMon local box wide control.		Package
Register Address: ED1H, 3793	MSR_C13_PMON_EVNTSEL0	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 0		Package
Register Address: ED2H, 3794	MSR_C13_PMON_EVNTSEL1	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 1		Package
Register Address: ED3H, 3795	MSR_C13_PMON_EVNTSEL2	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 2		Package
Register Address: ED4H, 3796	MSR_C13_PMON_EVNTSEL3	
Uncore C-Box 13 PerfMon Event Select for C-Box 13 Counter 3		Package
Register Address: ED5H, 3797	MSR_C13_PMON_BOX_FILTER0	
Uncore C-Box 13 PerfMon Box Wide Filter 0		Package
Register Address: ED6H, 3798	MSR_C13_PMON_BOX_FILTER1	
Uncore C-Box 13 PerfMon Box Wide Filter 1		Package
Register Address: ED7H, 3799	MSR_C13_PMON_BOX_STATUS	
Uncore C-Box 13 PerfMon Box Wide Status		Package
Register Address: ED8H, 3800	MSR_C13_PMON_CTR0	
Uncore C-Box 13 PerfMon Counter 0		Package
Register Address: ED9H, 3801	MSR_C13_PMON_CTR1	
Uncore C-Box 13 PerfMon Counter 1		Package
Register Address: EDAH, 3802	MSR_C13_PMON_CTR2	
Uncore C-Box 13 PerfMon Counter 2		Package
Register Address: EDBH, 3803	MSR_C13_PMON_CTR3	
Uncore C-Box 13 PerfMon Counter 3		Package
Register Address: EE0H, 3808	MSR_C14_PMON_BOX_CTL	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 14 PerfMon Local Box Wide Control		Package
Register Address: EE1H, 3809	MSR_C14_PMON_EVNTSEL0	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 0		Package
Register Address: EE2H, 3810	MSR_C14_PMON_EVNTSEL1	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 1		Package
Register Address: EE3H, 3811	MSR_C14_PMON_EVNTSEL2	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 2		Package
Register Address: EE4H, 3812	MSR_C14_PMON_EVNTSEL3	
Uncore C-Box 14 PerfMon Event Select for C-Box 14 Counter 3		Package
Register Address: EE5H, 3813	MSR_C14_PMON_BOX_FILTER	
Uncore C-Box 14 PerfMon Box Wide Filter 0		Package
Register Address: EE6H, 3814	MSR_C14_PMON_BOX_FILTER1	
Uncore C-Box 14 PerfMon Box Wide Filter 1		Package
Register Address: EE7H, 3815	MSR_C14_PMON_BOX_STATUS	
Uncore C-Box 14 PerfMon Box Wide Status		Package
Register Address: EE8H, 3816	MSR_C14_PMON_CTR0	
Uncore C-Box 14 PerfMon Counter 0		Package
Register Address: EE9H, 3817	MSR_C14_PMON_CTR1	
Uncore C-Box 14 PerfMon Counter 1		Package
Register Address: EEAH, 3818	MSR_C14_PMON_CTR2	
Uncore C-Box 14 PerfMon Counter 2		Package
Register Address: EEBH, 3819	MSR_C14_PMON_CTR3	
Uncore C-Box 14 PerfMon Counter 3		Package
Register Address: EF0H, 3824	MSR_C15_PMON_BOX_CTL	
Uncore C-Box 15 PerfMon Local Box Wide Control		Package
Register Address: EF1H, 3825	MSR_C15_PMON_EVNTSEL0	
Uncore C-Box 15 PerfMon Event Select for C-Box 15 Counter 0		Package
Register Address: EF2H, 3826	MSR_C15_PMON_EVNTSEL1	
Uncore C-Box 15 PerfMon Event Select for C-Box 15 Counter 1		Package
Register Address: EF3H, 3827	MSR_C15_PMON_EVNTSEL2	
Uncore C-Box 15 PerfMon Event Select for C-Box 15 Counter 2		Package
Register Address: EF4H, 3828	MSR_C15_PMON_EVNTSEL3	
Uncore C-Box 15 PerfMon Event Select for C-Box 15 Counter 3		Package
Register Address: EF5H, 3829	MSR_C15_PMON_BOX_FILTER0	
Uncore C-Box 15 PerfMon Box Wide Filter 0		Package
Register Address: EF6H, 3830	MSR_C15_PMON_BOX_FILTER1	
Uncore C-Box 15 PerfMon Box Wide Filter 1		Package

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: EF7H, 3831	MSR_C15_PMON_BOX_STATUS	
Uncore C-Box 15 PerfMon Box Wide Status		Package
Register Address: EF8H, 3832	MSR_C15_PMON_CTR0	
Uncore C-Box 15 PerfMon Counter 0		Package
Register Address: EF9H, 3833	MSR_C15_PMON_CTR1	
Uncore C-Box 15 PerfMon Counter 1		Package
Register Address: EFAH, 3834	MSR_C15_PMON_CTR2	
Uncore C-Box 15 PerfMon Counter 2		Package
Register Address: EFBH, 3835	MSR_C15_PMON_CTR3	
Uncore C-Box 15 PerfMon Counter 3		Package
Register Address: F00H, 3840	MSR_C16_PMON_BOX_CTL	
Uncore C-Box 16 PerfMon for Box-Wide Control		Package
Register Address: F01H, 3841	MSR_C16_PMON_EVNTSEL0	
Uncore C-Box 16 PerfMon Event Select for C-Box 16 Counter 0		Package
Register Address: F02H, 3842	MSR_C16_PMON_EVNTSEL1	
Uncore C-Box 16 PerfMon Event Select for C-Box 16 Counter 1		Package
Register Address: F03H, 3843	MSR_C16_PMON_EVNTSEL2	
Uncore C-Box 16 PerfMon Event Select for C-Box 16 Counter 2		Package
Register Address: F04H, 3844	MSR_C16_PMON_EVNTSEL3	
Uncore C-Box 16 PerfMon Event Select for C-Box 16 Counter 3		Package
Register Address: F05H, 3845	MSR_C16_PMON_BOX_FILTER0	
Uncore C-Box 16 PerfMon Box Wide Filter 0		Package
Register Address: F06H, 3846	MSR_C16_PMON_BOX_FILTER1	
Uncore C-Box 16 PerfMon Box Wide Filter 1		Package
Register Address: F07H, 3847	MSR_C16_PMON_BOX_STATUS	
Uncore C-Box 16 PerfMon Box Wide Status		Package
Register Address: F08H, 3848	MSR_C16_PMON_CTR0	
Uncore C-Box 16 PerfMon Counter 0		Package
Register Address: F09H, 3849	MSR_C16_PMON_CTR1	
Uncore C-Box 16 PerfMon Counter 1		Package
Register Address: F0AH, 3850	MSR_C16_PMON_CTR2	
Uncore C-Box 16 PerfMon Counter 2		Package
Register Address: F0BH, 3851	MSR_C16_PMON_CTR3	
Uncore C-Box 16 PerfMon Counter 3		Package
Register Address: F10H, 3856	MSR_C17_PMON_BOX_CTL	
Uncore C-Box 17 PerfMon for Box-Wide Control		Package
Register Address: F11H, 3857	MSR_C17_PMON_EVNTSEL0	

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 17 PerfMon Event Select for C-Box 17 Counter 0		Package
Register Address: F12H, 3858	MSR_C17_PMON_EVNTSEL1	
Uncore C-Box 17 PerfMon Event Select for C-Box 17 Counter 1		Package
Register Address: F13H, 3859	MSR_C17_PMON_EVNTSEL2	
Uncore C-Box 17 PerfMon Event Select for C-Box 17 Counter 2		Package
Register Address: F14H, 3860	MSR_C17_PMON_EVNTSEL3	
Uncore C-Box 17 PerfMon Event Select for C-Box 17 Counter 3		Package
Register Address: F15H, 3861	MSR_C17_PMON_BOX_FILTER0	
Uncore C-Box 17 PerfMon Box Wide Filter 0		Package
Register Address: F16H, 3862	MSR_C17_PMON_BOX_FILTER1	
Uncore C-Box 17 PerfMon Box Wide Filter1		Package
Register Address: F17H, 3863	MSR_C17_PMON_BOX_STATUS	
Uncore C-Box 17 PerfMon Box Wide Status		Package
Register Address: F18H, 3864	MSR_C17_PMON_CTR0	
Uncore C-Box 17 PerfMon Counter 0		Package
Register Address: F19H, 3865	MSR_C17_PMON_CTR1	
Uncore C-Box 17 PerfMon Counter 1		Package
Register Address: F1AH, 3866	MSR_C17_PMON_CTR2	
Uncore C-Box 17 PerfMon Counter 2		Package
Register Address: F1BH, 3867	MSR_C17_PMON_CTR3	
Uncore C-Box 17 PerfMon Counter 3		Package

2.15 MSRS IN THE INTEL® CORE™ M PROCESSORS AND THE 5TH GENERATION INTEL® CORE™ PROCESSORS

The Intel® Core™ M-5xxx processors, 5th generation Intel® Core™ Processors, and the Intel® Xeon® Processor E3-1200 v4 family are based on Broadwell microarchitecture. The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3DH. The Intel® Xeon® Processor E3-1200 v4 family and 5th generation Intel® Core™ Processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_47H. Processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3DH or 06_47H support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, Table 2-34, and Table 2-35. For an MSR listed in Table 2-35 that also appears in the model-specific tables of prior generations, Table 2-35 supersedes prior generation tables.

Table 2-34 lists MSRs that are common to processors based on the Broadwell microarchitectures (including CPUID Signature DisplayFamily_DisplayModel values of 06_3DH, 06_47H, 06_4FH, and 06_56H).

Table 2-34. Additional MSRs Common to Processors Based on Broadwell Microarchitectures

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	

Table 2-34. Additional MSRs Common to Processors Based on Broadwell Microarchitectures

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2 and Section 21.6.2.2, "Global Counter Control Facilities."		Thread
0	Ovf_PMC0	
1	Ovf_PMC1	
2	Ovf_PMC2	
3	Ovf_PMC3	
31:4	Reserved	
32	Ovf_FixedCtr0	
33	Ovf_FixedCtr1	
34	Ovf_FixedCtr2	
54:35	Reserved.	
55	Trace_ToPA_PMI See Section 34.2.7.2, "Table of Physical Addresses (ToPA)."	
60:56	Reserved.	
61	Ovf_Uncore	
62	Ovf_BufDSSAVE	
63	CondChgd	
Register Address: 390H, 912	IA32_PERF_GLOBAL_OVF_CTRL	
See Table 2-2 and Section 21.6.2.2, "Global Counter Control Facilities."		Thread
0	Set 1 to clear Ovf_PMC0.	
1	Set 1 to clear Ovf_PMC1.	
2	Set 1 to clear Ovf_PMC2.	
3	Set 1 to clear Ovf_PMC3.	
31:4	Reserved.	
32	Set 1 to clear Ovf_FixedCtr0.	
33	Set 1 to clear Ovf_FixedCtr1.	
34	Set 1 to clear Ovf_FixedCtr2	
54:35	Reserved.	
55	Set 1 to clear Trace_ToPA_PMI. See Section 34.2.7.2, "Table of Physical Addresses (ToPA)."	
60:56	Reserved.	
61	Set 1 to clear Ovf_Uncore.	
62	Set 1 to clear Ovf_BufDSSAVE.	
63	Set 1 to clear CondChgd.	
Register Address: 560H, 1376	IA32_RTIT_OUTPUT_BASE	
Trace Output Base Register (R/W)		Thread
6:0	Reserved.	
MAXPHYADDR[1]-1:7	Base physical address.	
63:MAXPHYADDR	Reserved.	

Table 2-34. Additional MSRs Common to Processors Based on Broadwell Microarchitectures

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 561H, 1377	IA32_RTIT_OUTPUT_MASK_PTRS	
Trace Output Mask Pointers Register (R/W)		Thread
6:0	Reserved.	
31:7	MaskOrTableOffset	
63:32	Output Offset.	
Register Address: 570H, 1392	IA32_RTIT_CTL	
Trace Control Register (R/W)		Thread
0	TraceEn	
1	Reserved, must be zero.	
2	OS	
3	User	
6:4	Reserved, must be zero.	
7	CR3Filter	
8	ToPA Writing 0 will #GP if also setting TraceEn.	
9	Reserved, must be zero.	
10	TSCEn	
11	DisRETC	
12	Reserved, must be zero.	
13	Reserved; writing 0 will #GP if also setting TraceEn.	
63:14	Reserved, must be zero.	
Register Address: 571H, 1393	IA32_RTIT_STATUS	
Tracing Status Register (R/W)		Thread
0	Reserved, writes ignored.	
1	ContexEn, writes ignored.	
2	TriggerEn, writes ignored.	
3	Reserved	
4	Error (R/W)	
5	Stopped	
63:6	Reserved, must be zero.	
Register Address: 572H, 1394	IA32_RTIT_CR3_MATCH	
Trace Filter CR3 Match Register (R/W)		Thread
4:0	Reserved.	
63:5	CR3[63:5] value to match.	
Register Address: 620H, 1568	MSR_UNCORE_RATIO_LIMIT	
Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.		Package

Table 2-34. Additional MSRs Common to Processors Based on Broadwell Microarchitectures

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
6:0	MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.	
7	Reserved.	
14:8	MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.	
63:15	Reserved.	

NOTES:

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

Table 2-35 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.		Core
3:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10	
9:4	Reserved.	
10	I/O MWAIT Redirection Enable (R/W)	
14:11	Reserved.	
15	CFG Lock (R/WO)	
24:16	Reserved.	
25	C3 State Auto Demotion Enable (R/W)	
26	C1 State Auto Demotion Enable (R/W)	
27	Enable C3 Undemotion (R/W)	

Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
28	Enable C1 Undemotion (R/W)	
29	Enable Package C-State Auto-Demotion (R/W)	
30	Enable Package C-State Undemotion (R/W)	
63:31	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
39:32	Maximum Ratio Limit for 5C Maximum turbo ratio limit of 5core active.	Package
47:40	Maximum Ratio Limit for 6C Maximum turbo ratio limit of 6core active.	Package
63:48	Reserved.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
See Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, and Table 2-34 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3DH.		

2.16 MSRS IN THE INTEL® XEON® PROCESSOR E5 V4 FAMILY

The MSRs listed in Table 2-36 are available and common to the Intel® Xeon® Processor D Product Family (CPUID Signature DisplayFamily_DisplayModel value of 06_56H) and to the Intel Xeon processors E5 v4 and E7 v4 families (CPUID Signature DisplayFamily_DisplayModel value of 06_4FH). These processors are based on Broadwell microarchitecture.

See Section 2.16.1 for lists of tables of MSRs that are supported by the Intel® Xeon® Processor D Family.

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 4EH, 78	IA32_PPIN_CTL (MSR_PPIN_CTL)	
Protected Processor Inventory Number Enable Control (R/W)		Package

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
0	LockOut (R/WO) See Table 2-2.	
1	Enable_PPIN (R/W) See Table 2-2.	
63:2	Reserved	
Register Address: 4FH, 79	IA32_PPIN (MSR_PPIN)	
Protected Processor Inventory Number (R/O)		Package
63:0	Protected Processor Inventory Number (R/O) See Table 2-2.	
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) See Table 2-26.	Package
22:16	Reserved.	
23	PPIN_CAP (R/O) See Table 2-26.	Package
27:24	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) See Table 2-26.	Package
29	Programmable TDP Limit for Turbo Mode (R/O) See Table 2-26.	Package
30	Programmable TJ OFFSET (R/O) See Table 2-26.	Package
39:31	Reserved.	
47:40	Maximum Efficiency Ratio (R/O) See Table 2-26.	Package
63:48	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.		Core

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W)	
14:11	Reserved.	
15	CFG Lock (R/WO)	
16	Automatic C-State Conversion Enable (R/W) If 1, the processor will convert HALT or MWAT(C1) to MWAIT(C6).	
24:17	Reserved.	
25	C3 State Auto Demotion Enable (R/W)	
26	C1 State Auto Demotion Enable (R/W)	
27	Enable C3 Undemotion (R/W)	
28	Enable C1 Undemotion (R/W)	
29	Package C State Demotion Enable (R/W)	
30	Package C State Undemotion Enable (R/W)	
63:31	Reserved.	
Register Address: 179H, 377	IA32_MCG_CAP	
Global Machine Check Capability (R/O)		Thread
7:0	Count	
8	MCG_CTL_P	
9	MCG_EXT_P	
10	MCP_CMCI_P	
11	MCG_TES_P	
15:12	Reserved	
23:16	MCG_EXT_CNT	
24	MCG_SER_P	
25	MCG_EM_P	
26	MCG_ELOG_P	
63:27	Reserved.	
Register Address: 17DH, 381	MSR_SMM_MCA_CAP	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.		Thread
57:0	Reserved.	
58	SMM_Code_Access_Chk (SMM-RO) If set to 1, indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.	
59	Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.	
63:60	Reserved.	
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Core
0	Thermal Status (R/O) See Table 2-2.	
1	Thermal Status Log (R/WC0) See Table 2-2.	
2	PROTCHOT # or FORCEPR# Status (R/O) See Table 2-2.	
3	PROTCHOT # or FORCEPR# Log (R/WC0) See Table 2-2.	
4	Critical Temperature Status (R/O) See Table 2-2.	
5	Critical Temperature Status Log (R/WC0) See Table 2-2.	
6	Thermal Threshold #1 Status (R/O) See Table 2-2.	
7	Thermal Threshold #1 Log (R/WC0) See Table 2-2.	
8	Thermal Threshold #2 Status (R/O) See Table 2-2.	
9	Thermal Threshold #2 Log (R/WC0) See Table 2-2.	
10	Power Limitation Status (R/O) See Table 2-2.	
11	Power Limitation Log (R/WC0) See Table 2-2.	
12	Current Limit Status (R/O) See Table 2-2.	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
13	Current Limit Log (R/WC0) See Table 2-2.	
14	Cross Domain Limit Status (R/O) See Table 2-2.	
15	Cross Domain Limit Log (R/WC0) See Table 2-2.	
22:16	Digital Readout (R/O) See Table 2-2.	
26:23	Reserved.	
30:27	Resolution in Degrees Celsius (R/O) See Table 2-2.	
31	Reading Valid (R/O) See Table 2-2.	
63:32	Reserved.	
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target		Package
15:0	Reserved.	
23:16	Temperature Target (R/O) See Table 2-26.	
27:24	TCC Activation Offset (R/W) See Table 2-26.	
63:28	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 1C	Package
15:8	Maximum Ratio Limit for 2C	Package
23:16	Maximum Ratio Limit for 3C	Package
31:24	Maximum Ratio Limit for 4C	Package
39:32	Maximum Ratio Limit for 5C	Package
47:40	Maximum Ratio Limit for 6C	Package
55:48	Maximum Ratio Limit for 7C	Package
63:56	Maximum Ratio Limit for 8C	Package
Register Address: 1AEH, 430	MSR_TURBO_RATIO_LIMIT1	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
7:0	Maximum Ratio Limit for 9C	Package
15:8	Maximum Ratio Limit for 10C	Package

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
23:16	Maximum Ratio Limit for 11C	Package
31:24	Maximum Ratio Limit for 12C	Package
39:32	Maximum Ratio Limit for 13C	Package
47:40	Maximum Ratio Limit for 14C	Package
55:48	Maximum Ratio Limit for 15C	Package
63:56	Maximum Ratio Limit for 16C	Package
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers Used in RAPL Interfaces (R/O)		Package
3:0	Power Units See Section 16.10.1, “RAPL Interfaces.”	Package
7:4	Reserved.	Package
12:8	Energy Status Units Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).	Package
15:13	Reserved.	Package
19:16	Time Units See Section 16.10.1, “RAPL Interfaces.”	Package
63:20	Reserved.	
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
DRAM RAPL Power Limit Control (R/W) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) Energy consumed by DRAM devices.		Package
31:0	Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).	
63:32	Reserved.	
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	
DRAM RAPL Parameters (R/W) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 620H, 1568	MSR_UNCORE_RATIO_LIMIT	
Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.		Package
63:15	Reserved.	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
14:8	MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.	
7	Reserved.	
6:0	MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
Reserved (R/O) Reads return 0.		Package
Register Address: 690H, 1680	MSR_CORE_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)		Package
0	PROCHOT Status (R0) When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.	
2	Power Budget Management Status (R0) When set, frequency is reduced below the operating system request due to PBM limit.	
3	Platform Configuration Services Status (R0) When set, frequency is reduced below the operating system request due to PCS limit.	
4	Reserved.	
5	Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.	
6	VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.	
7	Reserved.	
8	Electrical Design Point Status (R0) When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).	
9	Reserved.	
10	Multi-Core Turbo Status (R0) When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits.	
12:11	Reserved.	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
13	Core Frequency P1 Status (R0) When set, frequency is reduced below max non-turbo P1.	
14	Core Max N-Core Turbo Frequency Limiting Status (R0) When set, frequency is reduced below max n-core turbo frequency.	
15	Core Frequency Limiting Status (R0) When set, frequency is reduced below the operating system request.	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
18	Power Budget Management Log When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19	Platform Configuration Services Log When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
20	Reserved.	
21	Autonomous Utilization-Based Frequency Control Log When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	Reserved.	
24	Electrical Design Point Log When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Reserved.	
26	Multi-Core Turbo Log When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28:27	Reserved.	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
29	Core Frequency P1 Log When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
30	Core Max N-Core Turbo Frequency Limiting Log When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
31	Core Frequency Limiting Log When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:32	Reserved.	
Register Address: 770H, 1904	IA32_PM_ENABLE	
See Section 16.4.2, “Enabling HWP.”		Package
Register Address: 771H, 1905	IA32_HWP_CAPABILITIES	
See Section 16.4.3, “HWP Performance Range and Dynamic Capabilities.”		Thread
Register Address: 774H, 1908	IA32_HWP_REQUEST	
See Section 16.4.4, “Managing HWP.”		Thread
7:0	Minimum Performance (R/W)	
15:8	Maximum Performance (R/W)	
23:16	Desired Performance (R/W)	
63:24	Reserved.	
Register Address: 777H, 1911	IA32_HWP_STATUS	
See Section 16.4.5, “HWP Feedback.”		Thread
1:0	Reserved.	
2	Excursion to Minimum (R/O)	
63:3	Reserved.	
Register Address: C8DH, 3213	IA32_QM_EVTSEL	
Monitoring Event Select Register (R/W) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.		Thread
7:0	EventID (R/W) Event encoding: 0x00: No monitoring. 0x01: L3 occupancy monitoring. 0x02: Total memory bandwidth monitoring. 0x03: Local memory bandwidth monitoring. All other encoding reserved.	
31:8	Reserved.	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
41:32	RMID (R/W)	
63:42	Reserved.	
Register Address: C8FH, 3215	IA32_PQR_ASSOC	
Resource Association Register (R/W)		Thread
9:0	RMID	
31:10	Reserved.	
51:32	CLOS (R/W)	
63: 52	Reserved.	
Register Address: C90H, 3216	IA32_L3_QOS_MASK_0	
L3 Class Of Service Mask - CLOS 0 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 0 enforcement.	
63:20	Reserved.	
Register Address: C91H, 3217	IA32_L3_QOS_MASK_1	
L3 Class Of Service Mask - CLOS 1 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 1 enforcement.	
63:20	Reserved.	
Register Address: C92H, 3218	IA32_L3_QOS_MASK_2	
L3 Class Of Service Mask - CLOS 2 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 2 enforcement.	
63:20	Reserved.	
Register Address: C93H, 3219	IA32_L3_QOS_MASK_3	
L3 Class Of Service Mask - CLOS 3 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 3 enforcement.	
63:20	Reserved.	
Register Address: C94H, 3220	IA32_L3_QOS_MASK_4	
L3 Class Of Service Mask - CLOS 4 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 4 enforcement.	
63:20	Reserved.	
Register Address: C95H, 3221	IA32_L3_QOS_MASK_5	
L3 Class Of Service Mask - CLOS 5 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 5 enforcement.	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:20	Reserved.	
Register Address: C96H, 3222	IA32_L3_QOS_MASK_6	
L3 Class Of Service Mask - CLOS 6 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 6 enforcement.	
63:20	Reserved.	
Register Address: C97H, 3223	IA32_L3_QOS_MASK_7	
L3 Class Of Service Mask - CLOS 7 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 7 enforcement.	
63:20	Reserved.	
Register Address: C98H, 3224	IA32_L3_QOS_MASK_8	
L3 Class Of Service Mask - CLOS 8 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 8 enforcement.	
63:20	Reserved.	
Register Address: C99H, 3225	IA32_L3_QOS_MASK_9	
L3 Class Of Service Mask - CLOS 9 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 9 enforcement.	
63:20	Reserved.	
Register Address: C9AH, 3226	IA32_L3_QOS_MASK_10	
L3 Class Of Service Mask - CLOS 10 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 10 enforcement.	
63:20	Reserved.	
Register Address: C9BH, 3227	IA32_L3_QOS_MASK_11	
L3 Class Of Service Mask - CLOS 11 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=11.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 11 enforcement.	
63:20	Reserved.	
Register Address: C9CH, 3228	IA32_L3_QOS_MASK_12	
L3 Class Of Service Mask - CLOS 12 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=12.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 12 enforcement.	
63:20	Reserved.	
Register Address: C9DH, 3229	IA32_L3_QOS_MASK_13	

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
L3 Class Of Service Mask - CLOS 13 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=13.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 13 enforcement.	
63:20	Reserved.	
Register Address: C9EH, 3230	IA32_L3_QOS_MASK_14	
L3 Class Of Service Mask - CLOS 14 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 14 enforcement.	
63:20	Reserved.	
Register Address: C9FH, 3231	IA32_L3_QOS_MASK_15	
L3 Class Of Service Mask - CLOS 15 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 15 enforcement.	
63:20	Reserved.	

2.16.1 Additional MSRs Supported in the Intel® Xeon® Processor D Product Family

The MSRs listed in Table 2-37 are available to Intel® Xeon® Processor D Product Family (CPUID Signature DisplayFamily_DisplayModel value of 06_56H). The Intel® Xeon® processor D product family is based on Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-20, Table 2-29, Table 2-34, Table 2-36, and Table 2-37.

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with a CPUID Signature DisplayFamily_DisplayModel Value of 06_56H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1ACH, 428	MSR_TURBO_RATIO_LIMIT3	
Config Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
62:0	Reserved.	Package
63	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).	Package
Register Address: 286H, 646	IA32_MC6_CTL2	
See Table 2-2.		Package
Register Address: 287H, 647	IA32_MC7_CTL2	
See Table 2-2.		Package
Register Address: 289H, 649	IA32_MC9_CTL2	
See Table 2-2.		Package

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with a CPUID Signature DisplayFamily_DisplayModel Value of 06_56H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 28AH, 650	IA32_MC10_CTL2	
See Table 2-2.		Package
Register Address: 291H, 657	IA32_MC17_CTL2	
See Table 2-2.		Package
Register Address: 292H, 658	IA32_MC18_CTL2	
See Table 2-2.		Package
Register Address: 293H, 659	IA32_MC19_CTL2	
See Table 2-2.		Package
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41BH, 1051	IA32_MC6_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41CH, 1052	IA32_MC7_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41DH, 1053	IA32_MC7_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41EH, 1054	IA32_MC7_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41FH, 1055	IA32_MC7_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 424H, 1060	IA32_MC9_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 425H, 1061	IA32_MC9_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with a CPUID Signature DisplayFamily_DisplayModel Value of 06_56H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 426H, 1062	IA32_MC9_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 427H, 1063	IA32_MC9_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 428H, 1064	IA32_MC10_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 429H, 1065	IA32_MC10_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42AH, 1066	IA32_MC10_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42BH, 1067	IA32_MC10_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 444H, 1092	IA32_MC17_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 445H, 1093	IA32_MC17_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 446H, 1094	IA32_MC17_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 447H, 1095	IA32_MC17_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 448H, 1096	IA32_MC18_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 449H, 1097	IA32_MC18_STATUS	

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with a CPUID Signature DisplayFamily_DisplayModel Value of 06_56H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44AH, 1098	IA32_MC18_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44BH, 1099	IA32_MC18_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44CH, 1100	IA32_MC19_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44DH, 1101	IA32_MC19_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44EH, 1102	IA32_MC19_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44FH, 1103	IA32_MC19_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
See Table 2-20, Table 2-29, Table 2-34, and Table 2-36 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_56H.		

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.16.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families

The MSRs listed in Table 2-37 are available to the Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID Signature DisplayFamily_DisplayModel value of 06_4FH). The Intel® Xeon® processor E5 v4 family is based on Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-34, Table 2-36, and Table 2-38.

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1ACH, 428	MSR_TURBO_RATIO_LIMIT3	
Config Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.		Package
62:0	Reserved.	Package
63	Semaphore for Turbo Ratio Limit Configuration If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1, and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).	Package
Register Address: 285H, 645	IA32_MC5_CTL2	
See Table 2-2.		Package
Register Address: 286H, 646	IA32_MC6_CTL2	
See Table 2-2.		Package
Register Address: 287H, 647	IA32_MC7_CTL2	
See Table 2-2.		Package
Register Address: 288H, 648	IA32_MC8_CTL2	
See Table 2-2.		Package
Register Address: 289H, 649	IA32_MC9_CTL2	
See Table 2-2.		Package
Register Address: 28AH, 650	IA32_MC10_CTL2	
See Table 2-2.		Package
Register Address: 28BH, 651	IA32_MC11_CTL2	
See Table 2-2.		Package
Register Address: 28CH, 652	IA32_MC12_CTL2	
See Table 2-2.		Package
Register Address: 28DH, 653	IA32_MC13_CTL2	
See Table 2-2.		Package
Register Address: 28EH, 654	IA32_MC14_CTL2	
See Table 2-2.		Package
Register Address: 28FH, 655	IA32_MC15_CTL2	
See Table 2-2.		Package
Register Address: 290H, 656	IA32_MC16_CTL2	
See Table 2-2.		Package
Register Address: 291H, 657	IA32_MC17_CTL2	
See Table 2-2.		Package
Register Address: 292H, 658	IA32_MC18_CTL2	
See Table 2-2.		Package

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 293H, 659	IA32_MC19_CTL2	
See Table 2-2.		Package
Register Address: 294H, 660	IA32_MC20_CTL2	
See Table 2-2.		Package
Register Address: 295H, 661	IA32_MC21_CTL2	
See Table 2-2.		Package
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI 0 module.		Package
Register Address: 415H, 1045	IA32_MC5_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI 0 module.		Package
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI 0 module.		Package
Register Address: 417H, 1047	IA32_MC5_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI 0 module.		Package
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41BH, 1051	IA32_MC6_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41CH, 1052	IA32_MC7_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41DH, 1053	IA32_MC7_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41EH, 1054	IA32_MC7_ADDR	

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 41FH, 1055	IA32_MC7_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the home agent HA 0.		Package
Register Address: 420H, 1056	IA32_MC8_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the home agent HA 1.		Package
Register Address: 421H, 1057	IA32_MC8_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the home agent HA 1.		Package
Register Address: 422H, 1058	IA32_MC8_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the home agent HA 1.		Package
Register Address: 423H, 1059	IA32_MC8_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the home agent HA 1.		Package
Register Address: 424H, 1060	IA32_MC9_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 425H, 1061	IA32_MC9_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 426H, 1062	IA32_MC9_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 427H, 1063	IA32_MC9_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 428H, 1064	IA32_MC10_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 429H, 1065	IA32_MC10_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42AH, 1066	IA32_MC10_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 42BH, 1067	IA32_MC10_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42CH, 1068	IA32_MC11_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42DH, 1069	IA32_MC11_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42EH, 1070	IA32_MC11_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 42FH, 1071	IA32_MC11_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 430H, 1072	IA32_MC12_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 431H, 1073	IA32_MC12_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 432H, 1074	IA32_MC12_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 433H, 1075	IA32_MC12_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 434H, 1076	IA32_MC13_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 435H, 1077	IA32_MC13_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 436H, 1078	IA32_MC13_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 437H, 1079	IA32_MC13_MISC	

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 438H, 1080	IA32_MC14_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 439H, 1081	IA32_MC14_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43AH, 1082	IA32_MC14_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43BH, 1083	IA32_MC14_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43CH, 1084	IA32_MC15_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43DH, 1085	IA32_MC15_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43EH, 1086	IA32_MC15_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 43FH, 1087	IA32_MC15_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 440H, 1088	IA32_MC16_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 441H, 1089	IA32_MC16_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 442H, 1090	IA32_MC16_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package
Register Address: 443H, 1091	IA32_MC16_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.		Package

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 444H, 1092	IA32_MC17_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 445H, 1093	IA32_MC17_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 446H, 1094	IA32_MC17_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 447H, 1095	IA32_MC17_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.		Package
Register Address: 448H, 1096	IA32_MC18_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 449H, 1097	IA32_MC18_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44AH, 1098	IA32_MC18_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44BH, 1099	IA32_MC18_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.		Package
Register Address: 44CH, 1100	IA32_MC19_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44DH, 1101	IA32_MC19_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44EH, 1102	IA32_MC19_ADDR	

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 44FH, 1103	IA32_MC19_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.		Package
Register Address: 450H, 1104	IA32_MC20_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 451H, 1105	IA32_MC20_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 452H, 1106	IA32_MC20_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 453H, 1107	IA32_MC20_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC20 reports MC errors from the Intel QPI 1 module.		Package
Register Address: 454H, 1108	IA32_MC21_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: 455H, 1109	IA32_MC21_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: 456H, 1110	IA32_MC21_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: 457H, 1111	IA32_MC21_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC21 reports MC errors from the Intel QPI 2 module.		Package
Register Address: C81H, 3201	IA32_L3_QOS_CFG	
Cache Allocation Technology Configuration (R/W)		Package
0	CAT Enable. Set 1 to enable Cache Allocation Technology.	
63:1	Reserved.	
See Table 2-20, Table 2-21, Table 2-29, and Table 2-30 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_45H.		

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.17 MSRS IN THE 6TH—13TH GENERATION INTEL® CORE™ PROCESSORS, 1ST—5TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILIES, INTEL® CORE™ ULTRA 7 PROCESSORS, 8TH GENERATION INTEL® CORE™ I3 PROCESSORS, INTEL® XEON® E PROCESSORS, INTEL® XEON® 6 P-CORE PROCESSORS, INTEL® XEON® 6 E-CORE PROCESSORS, AND INTEL® SERIES 2 CORE™ ULTRA PROCESSORS

6th generation Intel® Core™ processors are based on Skylake microarchitecture and have a CPUID Signature DisplayFamily_DisplayModel value of 06_4EH or 06_5EH.

The Intel® Xeon® Scalable Processor Family based on the Skylake microarchitecture, the 2nd generation Intel® Xeon® Scalable Processor Family based on the Cascade Lake product, and the 3rd generation Intel® Xeon® Scalable Processor Family based on the Cooper Lake product all have a CPUID Signature DisplayFamily_DisplayModel value of 06_55H.

7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture, 8th generation and 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on Coffee Lake microarchitecture; these processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_8EH or 06_9EH.

8th generation Intel® Core™ i3 processors are based on Cannon Lake microarchitecture and have a CPUID Signature DisplayFamily_DisplayModel value of 06_66H.

10th generation Intel® Core™ processors are based on Comet Lake microarchitecture (with a CPUID Signature DisplayFamily_DisplayModel value of 06_A5H or 06_A6H) and Ice Lake microarchitecture (with a CPUID Signature DisplayFamily_DisplayModel value of 06_7DH or 06_7EH).

11th generation Intel® Core™ processors are based on Tiger Lake microarchitecture and have a CPUID Signature DisplayFamily_DisplayModel value of 06_8CH or 06_8DH.

The 3rd generation Intel® Xeon® Scalable Processor Family is based on Ice Lake microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_6AH or 06_6CH.

12th generation Intel® Core™ processors supporting the Alder Lake performance hybrid architecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_97H or 06_9AH.

13th generation Intel® Core™ processors supporting the Raptor Lake performance hybrid architecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_BAH, 06_B7H, or 06_BFH.

The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_8FH.

The 5th generation Intel® Xeon® Scalable Processor Family is based on Emerald Rapids microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_CFH.

The Intel® Core™ Ultra 7 processors supporting the Meteor Lake hybrid architecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_AAH.

The Intel® Xeon® 6 P-core processor is based on the Granite Rapids microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_ADH or 06_AEH.

The Intel® Xeon® 6 E-core processor is based on the Sierra Forest microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_AFH.

The Intel® Series 2 Core™ Ultra processors supporting the Lunar Lake performance hybrid architecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_BDH.

These processors support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-25, Table 2-29, Table 2-35, and Table 2-39[1]. For an MSR listed in Table 2-39 that also appears in the model-specific tables of prior generations, Table 2-39 supersedes prior generation tables.

Tables 2-40 through 2-60 list additional supported MSR interfaces introduced in specific processors; see each table for additional details.

The notation of "Platform" in the Scope column (with respect to MSR_PLATFORM_ENERGY_COUNTER and MSR_PLATFORM_POWER_LIMIT) is limited to the power-delivery domain and the specifics of the power delivery integration may vary by platform vendor's implementation.

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64 Processor (R/W) See Table 2-2.		Thread
Register Address: FEH, 254	IA32_MTRRCAP	
MTRR Capability (R/O, Architectural) See Table 2-2		Thread
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Core
0	Thermal Status (R/O) See Table 2-2.	
1	Thermal Status Log (R/WC0) See Table 2-2.	
2	PROTCHOT # or FORCEPR# Status (R/O) See Table 2-2.	
3	PROTCHOT # or FORCEPR# Log (R/WC0) See Table 2-2.	
4	Critical Temperature Status (R/O) See Table 2-2.	
5	Critical Temperature Status Log (R/WC0) See Table 2-2.	
6	Thermal threshold #1 Status (R/O) See Table 2-2.	
7	Thermal threshold #1 Log (R/WC0) See Table 2-2.	
8	Thermal Threshold #2 Status (R/O) See Table 2-2.	

1. MSRs at the following addresses are not supported in the 12th generation Intel Core processor E-core: 3F7H. MSRs at the following addresses are not supported in the 12th generation Intel Core processor E-core or P-core: 652H, 653H, 655H, 656H, DB0H, DB1H, DB2H, and D90H.

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
9	Thermal Threshold #2 Log (R/WC0) See Table 2-2.	
10	Power Limitation Status (R/O) See Table 2-2.	
11	Power Limitation Log (R/WC0) See Table 2-2.	
12	Current Limit Status (R/O) See Table 2-2.	
13	Current Limit Log (R/WC0) See Table 2-2.	
14	Cross Domain Limit Status (R/O) See Table 2-2.	
15	Cross Domain Limit Log (R/WC0) See Table 2-2.	
22:16	Digital Readout (R/O) See Table 2-2.	
26:23	Reserved.	
30:27	Resolution in Degrees Celsius (R/O) See Table 2-2.	
31	Reading Valid (R/O) See Table 2-2.	
63:32	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1		Package
7:0	Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.	Package
15:8	Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.	Package
23:16	Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.	Package
31:24	Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.	Package
63:32	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.		Thread

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1FCH, 508	MSR_POWER_CTL	
Power Control Register See http://biosbits.org.		Core
0	Reserved.	
1	C1E Enable (R/W) When set to ‘1’, will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).	Package
18:2	Reserved.	
19	Disable Energy Efficiency Optimization (R/W) Setting this bit disables the P-States energy efficiency optimization. Default value is 0. Disable/enable the energy efficiency optimization in P-State legacy mode (when IA32_PM_ENABLE[HWP_ENABLE] = 0), has an effect only in the turbo range or into PERF_MIN_CTL value if it is not zero set. In HWP mode (IA32_PM_ENABLE[HWP_ENABLE] == 1), has an effect between the OS desired or OS maximize to the OS minimize performance setting.	
20	Disable Race to Halt Optimization (R/W) Setting this bit disables the Race to Halt optimization and avoids this optimization limitation to execute below the most efficient frequency ratio. Default value is 0 for processors that support Race to Halt optimization.	
63:21	Reserved.	
Register Address: 300H, 768	MSR_SGXOWNEREPOCH0	
Lower 64 Bit CR_SGXOWNEREPOCH (W) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.		Package
63:0	Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.	
Register Address: 301H, 769	MSR_SGXOWNEREPOCH1	
Upper 64 Bit CR_SGXOWNEREPOCH (W) Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.		Package
63:0	Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.	
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2 and Section 21.2.4, “Architectural Performance Monitoring Version 4.”		
0	Ovf_PMC0	Thread
1	Ovf_PMC1	Thread
2	Ovf_PMC2	Thread
3	Ovf_PMC3	Thread
4	Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)	Thread
5	Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)	Thread

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
6	Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)	Thread
7	Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)	Thread
31:8	Reserved.	
32	Ovf_FixedCtr0	Thread
33	Ovf_FixedCtr1	Thread
34	Ovf_FixedCtr2	Thread
54:35	Reserved	
55	Trace_ToPA_PMI	Thread
57:56	Reserved.	
58	LBR_Frz	Thread
59	CTR_Frz	Thread
60	ASCI	Thread
61	Ovf_Uncore	Thread
62	Ovf_BufDSSAVE	Thread
63	CondChgd	Thread
Register Address: 390H, 912	IA32_PERF_GLOBAL_STATUS_RESET	
See Table 2-2 and Section 21.2.4, "Architectural Performance Monitoring Version 4."		
0	Set 1 to clear Ovf_PMC0.	Thread
1	Set 1 to clear Ovf_PMC1.	Thread
2	Set 1 to clear Ovf_PMC2.	Thread
3	Set 1 to clear Ovf_PMC3.	Thread
4	Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4).	Thread
5	Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5).	Thread
6	Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6).	Thread
7	Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7).	Thread
31:8	Reserved.	
32	Set 1 to clear Ovf_FixedCtr0.	Thread
33	Set 1 to clear Ovf_FixedCtr1.	Thread
34	Set 1 to clear Ovf_FixedCtr2.	Thread
54:35	Reserved.	
55	Set 1 to clear Trace_ToPA_PMI.	Thread
57:56	Reserved.	
58	Set 1 to clear LBR_Frz.	Thread
59	Set 1 to clear CTR_Frz.	Thread
60	Set 1 to clear ASCI.	Thread

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
61	Set 1 to clear Ovf_Uncore.	Thread
62	Set 1 to clear Ovf_BufDSSAVE.	Thread
63	Set 1 to clear CondChgd.	Thread
Register Address: 391H, 913	IA32_PERF_GLOBAL_STATUS_SET	
See Table 2-2 and Section 21.2.4, “Architectural Performance Monitoring Version 4.”		
0	Set 1 to cause Ovf_PMC0 = 1.	Thread
1	Set 1 to cause Ovf_PMC1 = 1.	Thread
2	Set 1 to cause Ovf_PMC2 = 1.	Thread
3	Set 1 to cause Ovf_PMC3 = 1.	Thread
4	Set 1 to cause Ovf_PMC4=1 (if CPUID.0AH:EAX[15:8] > 4).	Thread
5	Set 1 to cause Ovf_PMC5=1 (if CPUID.0AH:EAX[15:8] > 5).	Thread
6	Set 1 to cause Ovf_PMC6=1 (if CPUID.0AH:EAX[15:8] > 6).	Thread
7	Set 1 to cause Ovf_PMC7=1 (if CPUID.0AH:EAX[15:8] > 7).	Thread
31:8	Reserved.	
32	Set 1 to cause Ovf_FixedCtr0 = 1.	Thread
33	Set 1 to cause Ovf_FixedCtr1 = 1.	Thread
34	Set 1 to cause Ovf_FixedCtr2 = 1.	Thread
54:35	Reserved.	
55	Set 1 to cause Trace_ToPA_PMI = 1.	Thread
57:56	Reserved.	
58	Set 1 to cause LBR_Frz = 1.	Thread
59	Set 1 to cause CTR_Frz = 1.	Thread
60	Set 1 to cause ASCI = 1.	Thread
61	Set 1 to cause Ovf_Uncore.	Thread
62	Set 1 to cause Ovf_BufDSSAVE.	Thread
63	Reserved.	
Register Address: 392H, 914	IA32_PERF_GLOBAL_INUSE	
See Table 2-2.		Thread
Register Address: 3F7H, 1015	MSR_PEBS_FRONTEND	
FrontEnd Precise Event Condition Select (R/W)		Thread
2:0	Event Code Select	
3	Reserved	
4	Event Code Select High	
7:5	Reserved.	
19:8	IDQ_Bubble_Length Specifier	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
22:20	IDQ_Bubble_Width Specifier	
63:23	Reserved.	
Register Address: 500H, 1280	IA32_SGX_SVN_STATUS	
Status and SVN Threshold of SGX Support for ACM (R/O)		Thread
0	Lock See Section 40.11.3, “Interactions with Authenticated Code Modules (ACMs).”	
15:1	Reserved.	
23:16	SGX_SVN_SINIT See Section 40.11.3, “Interactions with Authenticated Code Modules (ACMs).”	
63:24	Reserved.	
Register Address: 560H, 1376	IA32_RTIT_OUTPUT_BASE	
Trace Output Base Register (R/W) See Table 2-2.		Thread
Register Address: 561H, 1377	IA32_RTIT_OUTPUT_MASK_PTRS	
Trace Output Mask Pointers Register (R/W) See Table 2-2.		Thread
Register Address: 570H, 1392	IA32_RTIT_CTL	
Trace Control Register (R/W)		Thread
0	TraceEn	
1	CYCEn	
2	OS	
3	User	
6:4	Reserved, must be zero.	
7	CR3Filter	
8	ToPA Writing 0 will #GP if also setting TraceEn.	
9	MTCEn	
10	TSCEn	
11	DisRETC	
12	Reserved, must be zero.	
13	BranchEn	
17:14	MTCFreq	
18	Reserved, must be zero.	
22:19	CycThresh	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
23	Reserved, must be zero.	
27:24	PSBFreq	
31:28	Reserved, must be zero.	
35:32	ADDR0_CFG	
39:36	ADDR1_CFG	
63:40	Reserved, must be zero.	
Register Address: 571H, 1393	IA32_RTIT_STATUS	
Tracing Status Register (R/W)		Thread
0	FilterEn, writes ignored.	
1	ContexEn, writes ignored.	
2	TriggerEn, writes ignored.	
3	Reserved	
4	Error (R/W)	
5	Stopped	
31:6	Reserved, must be zero.	
48:32	PacketByteCnt	
63:49	Reserved, must be zero.	
Register Address: 572H, 1394	IA32_RTIT_CR3_MATCH	
Trace Filter CR3 Match Register (R/W)		Thread
4:0	Reserved	
63:5	CR3[63:5] value to match	
Register Address: 580H, 1408	IA32_RTIT_ADDR0_A	
Region 0 Start Address (R/W)		Thread
63:0	See Table 2-2.	
Register Address: 581H, 1409	IA32_RTIT_ADDR0_B	
Region 0 End Address (R/W)		Thread
63:0	See Table 2-2.	
Register Address: 582H, 1410	IA32_RTIT_ADDR1_A	
Region 1 Start Address (R/W)		Thread
63:0	See Table 2-2.	
Register Address: 583H, 1411	IA32_RTIT_ADDR1_B	
Region 1 End Address (R/W)		Thread
63:0	See Table 2-2.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
PP0 Energy Status (R/O) See Section 16.10.4, "PP0/PP1 RAPL Domains."		Package
Register Address: 64DH, 1613	MSR_PLATFORM_ENERGY_COUNTER	
Platform Energy Counter (R/O) This MSR is valid only if both platform vendor hardware implementation and BIOS enablement support it. This MSR will read 0 if not valid.		Platform
31:0	Total energy consumed by all devices in the platform that receive power from integrated power delivery mechanism, included platform devices are processor cores, SOC, memory, add-on or peripheral devices that get powered directly from the platform power delivery means. The energy units are specified in the MSR_RAPL_POWER_UNIT.Enery_Status_Unit.	
63:32	Reserved.	
Register Address: 64EH, 1614	MSR_PPERF	
Productive Performance Count (R/O)		Thread
63:0	Hardware's view of workload scalability. See Section 16.4.5.1.	
Register Address: 64FH, 1615	MSR_CORE_PERF_LIMIT_REASONS	
	Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)	Package
0	PROCHOT Status (R0) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.	
3:2	Reserved.	
4	Residency State Regulation Status (R0) When set, frequency is reduced below the operating system request due to residency state regulation limit.	
5	Running Average Thermal Limit Status (R0) When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).	
6	VR Therm Alert Status (R0) When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).	
7	VR Therm Design Current Status (R0) When set, frequency is reduced below the operating system request due to VR thermal design current limit.	
8	Other Status (R0) When set, frequency is reduced below the operating system request due to electrical or other constraints.	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
9	Reserved.	
10	Package/Platform-Level Power Limiting PL1 Status (R0) When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.	
11	Package/Platform-Level PL2 Power Limiting Status (R0) When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.	
12	Max Turbo Limit Status (R0) When set, frequency is reduced below the operating system request due to multi-core turbo limits.	
13	Turbo Transition Attenuation Status (R0) When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.	
15:14	Reserved.	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19:18	Reserved.	
20	Residency State Regulation Log When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
21	Running Average Thermal Limit Log When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	VR Thermal Design Current Log When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
24	Other Log When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Reserved.	
26	Package/Platform-Level PL1 Power Limiting Log When set, indicates that the Package or Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
27	Package/Platform-Level PL2 Power Limiting Log When set, indicates that the Package or Platform Level PL2/PL3 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	Max Turbo Limit Log When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
29	Turbo Transition Attenuation Log When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:30	Reserved.	
Register Address: 652H, 1618	MSR_PKG_HDC_CONFIG	
HDC Configuration (R/W)		Package
2:0	PKG_Cx_Monitor Configures Package Cx state threshold for MSR_PKG_HDC_DEEP_RESIDENCY.	
63: 3	Reserved.	
Register Address: 653H, 1619	MSR_CORE_HDC_RESIDENCY	
Core HDC Idle Residency (R/O)		Core
63:0	Core_Cx_Duty_Cycle_Cnt	
Register Address: 655H, 1621	MSR_PKG_HDC_SHALLOW_RESIDENCY	
Accumulate the cycles the package was in C2 state and at least one logical processor was in forced idle (R/O)		Package
63:0	Pkg_C2_Duty_Cycle_Cnt	
Register Address: 656H, 1622	MSR_PKG_HDC_DEEP_RESIDENCY	
Package Cx HDC Idle Residency (R/O)		Package
63:0	Pkg_Cx_Duty_Cycle_Cnt	
Register Address: 658H, 1624	MSR_WEIGHTED_CORE_C0	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Core-count Weighted C0 Residency (R/O)		Package
63:0	Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.	
Register Address: 659H, 1625	MSR_ANY_CORE_C0	
Any Core C0 Residency (R/O)		Package
63:0	Increment at the same rate as the TSC. The increment each cycle is one if any processor core in the package is in C0.	
Register Address: 65AH, 1626	MSR_ANY_GFXE_C0	
Any Graphics Engine C0 Residency (R/O)		Package
63:0	Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.	
Register Address: 65BH, 1627	MSR_CORE_GFXE_OVERLAP_C0	
Core and Graphics Engine Overlapped C0 Residency (R/O)		Package
63:0	Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.	
Register Address: 65CH, 1628	MSR_PLATFORM_POWER_LIMIT	
Platform Power Limit Control (R/W-L) Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.		Platform
14:0	Platform Power Limit #1 Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field. The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT.	
15	Enable Platform Power Limit #1 When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #1 over the time window specified by Power Limit #1 Time Window.	
16	Platform Clamping Limitation #1 When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #1 value. This bit is writeable only when CPUID (EAX=6):EAX[4] is set.	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
23:17	Time Window for Platform Power Limit #1 Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation: Time Window = (float) ((1+(X/4))*(2^Y)), where: X = POWER_LIMIT_1_TIME[23:22] Y = POWER_LIMIT_1_TIME[21:17] The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is 0DH, and the unit is specified in MSR_RAPL_POWER_UNIT[Time Unit].	
31:24	Reserved.	
46:32	Platform Power Limit #2 Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor. The recommended default value is 1.25 times the Long Duration Power Limit (i.e., Platform Power Limit # 1).	
47	Enable Platform Power Limit #2 When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #2 over the Short Duration time window.	
48	Platform Clamping Limitation #2 When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.	
62:49	Reserved.	
63	Lock. Setting this bit will lock all other bits of this MSR until system RESET.	
Register Address: 690H, 1680	MSR_LASTBRANCH_16_FROM_IP	
Last Branch Record 16 From IP (R/W) One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.12.		Thread
Register Address: 691H, 1681	MSR_LASTBRANCH_17_FROM_IP	
Last Branch Record 17 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 692H, 1682	MSR_LASTBRANCH_18_FROM_IP	
Last Branch Record 18 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 693H, 1683	MSR_LASTBRANCH_19_FROM_IP	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 19From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 694H, 1684	MSR_LASTBRANCH_20_FROM_IP	
Last Branch Record 20 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 695H, 1685	MSR_LASTBRANCH_21_FROM_IP	
Last Branch Record 21 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 696H, 1686	MSR_LASTBRANCH_22_FROM_IP	
Last Branch Record 22 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 697H, 1687	MSR_LASTBRANCH_23_FROM_IP	
Last Branch Record 23 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 698H, 1688	MSR_LASTBRANCH_24_FROM_IP	
Last Branch Record 24 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 699H, 1689	MSR_LASTBRANCH_25_FROM_IP	
Last Branch Record 25 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 69AH, 1690	MSR_LASTBRANCH_26_FROM_IP	
Last Branch Record 26 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 69BH, 1691	MSR_LASTBRANCH_27_FROM_IP	
Last Branch Record 27 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 69CH, 1692	MSR_LASTBRANCH_28_FROM_IP	
Last Branch Record 28 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 69DH, 1693	MSR_LASTBRANCH_29_FROM_IP	
Last Branch Record 29 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 69EH, 1694	MSR_LASTBRANCH_30_FROM_IP	
Last Branch Record 30 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 69FH, 1695	MSR_LASTBRANCH_31_FROM_IP	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 31 From IP (R/W) See description of MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 6B0H, 1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in the Processor Graphics (R/W) (Frequency refers to processor graphics frequency.)		Package
0	PROCHOT Status (R0) When set, frequency is reduced due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced due to a thermal event.	
4:2	Reserved.	
5	Running Average Thermal Limit Status (R0) When set, frequency is reduced due to running average thermal limit.	
6	VR Therm Alert Status (R0) When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.	
7	VR Thermal Design Current Status (R0) When set, frequency is reduced due to VR TDC limit.	
8	Other Status (R0) When set, frequency is reduced due to electrical or other constraints.	
9	Reserved.	
10	Package/Platform-Level Power Limiting PL1 Status (R0) When set, frequency is reduced due to package/platform-level power limiting PL1.	
11	Package/Platform-Level PL2 Power Limiting Status (R0) When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.	
12	Inefficient Operation Status (R0) When set, processor graphics frequency is operating below target frequency.	
15:13	Reserved.	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
20:18	Reserved.	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
21	Running Average Thermal Limit Log When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	VR Thermal Design Current Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
24	Other Log When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Reserved.	
26	Package/Platform-Level PL1 Power Limiting Log When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
27	Package/Platform-Level PL2 Power Limiting Log When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	Inefficient Operation Log When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:29	Reserved.	
Register Address: 6B1H, 1713	MSR_RING_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in the Ring Interconnect (R/W) (Frequency refers to ring interconnect in the uncore.)		Package
0	PROCHOT Status (R0) When set, frequency is reduced due to assertion of external PROCHOT.	
1	Thermal Status (R0) When set, frequency is reduced due to a thermal event.	
4:2	Reserved.	
5	Running Average Thermal Limit Status (R0) When set, frequency is reduced due to running average thermal limit.	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
6	VR Therm Alert Status (R0) When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.	
7	VR Thermal Design Current Status (R0) When set, frequency is reduced due to VR TDC limit.	
8	Other Status (R0) When set, frequency is reduced due to electrical or other constraints.	
9	Reserved.	
10	Package/Platform-Level Power Limiting PL1 Status (R0) When set, frequency is reduced due to package/Platform-level power limiting PL1.	
11	Package/Platform-Level PL2 Power Limiting Status (R0) When set, frequency is reduced due to package/Platform-level power limiting PL2/PL3.	
15:12	Reserved	
16	PROCHOT Log When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
20:18	Reserved.	
21	Running Average Thermal Limit Log When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR Therm Alert Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	VR Thermal Design Current Log When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
24	Other Log When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
25	Reserved.	
26	Package/Platform-Level PL1 Power Limiting Log When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
27	Package/Platform-Level PL2 Power Limiting Log When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:28	Reserved.	
Register Address: 6D0H, 1744	MSR_LASTBRANCH_16_TO_IP	
Last Branch Record 16 To IP (R/W) One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.12.		Thread
Register Address: 6D1H, 1745	MSR_LASTBRANCH_17_TO_IP	
Last Branch Record 17 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D2H, 1746	MSR_LASTBRANCH_18_TO_IP	
Last Branch Record 18 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D3H, 1747	MSR_LASTBRANCH_19_TO_IP	
Last Branch Record 19To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D4H, 1748	MSR_LASTBRANCH_20_TO_IP	
Last Branch Record 20 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D5H, 1749	MSR_LASTBRANCH_21_TO_IP	
Last Branch Record 21 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D6H, 1750	MSR_LASTBRANCH_22_TO_IP	
Last Branch Record 22 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D7H, 1751	MSR_LASTBRANCH_23_TO_IP	
Last Branch Record 23 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D8H, 1752	MSR_LASTBRANCH_24_TO_IP	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 24 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6D9H, 1753	MSR_LASTBRANCH_25_TO_IP	
Last Branch Record 25 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6DAH, 1754	MSR_LASTBRANCH_26_TO_IP	
Last Branch Record 26 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6DBH, 1755	MSR_LASTBRANCH_27_TO_IP	
Last Branch Record 27 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6DCH, 1756	MSR_LASTBRANCH_28_TO_IP	
Last Branch Record 28 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6DDH, 1757	MSR_LASTBRANCH_29_TO_IP	
Last Branch Record 29 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6DEH, 1758	MSR_LASTBRANCH_30_TO_IP	
Last Branch Record 30 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 6DFH, 1759	MSR_LASTBRANCH_31_TO_IP	
Last Branch Record 31 To IP (R/W) See description of MSR_LASTBRANCH_0_TO_IP.		Thread
Register Address: 770H, 1904	IA32_PM_ENABLE	
See Section 16.4.2, “Enabling HWP.”		Package
Register Address: 771H, 1905	IA32_HWP_CAPABILITIES	
See Section 16.4.3, “HWP Performance Range and Dynamic Capabilities.”		Thread
Register Address: 772H, 1906	IA32_HWP_REQUEST_PKG	
See Section 16.4.4, “Managing HWP.”		Package
Register Address: 773H, 1907	IA32_HWP_INTERRUPT	
See Section 16.4.6, “HWP Notifications.”		Thread
Register Address: 774H, 1908	IA32_HWP_REQUEST	
See Section 16.4.4, “Managing HWP.”		Thread
7:0	Minimum Performance (R/W)	
15:8	Maximum Performance (R/W)	
23:16	Desired Performance (R/W)	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:24	Energy/Performance Preference (R/W)	
41:32	Activity Window (R/W)	
42	Package Control (R/W)	
63:43	Reserved.	
Register Address: 777H, 1911	IA32_HWP_STATUS	
See Section 16.4.5, “HWP Feedback.”		Thread
Register Address: D90H, 3472	IA32_BNDCFGS	
See Table 2-2.		Thread
Register Address: DA0H, 3488	IA32_XSS	
See Table 2-2.		Thread
Register Address: DB0H, 3504	IA32_PKG_HDC_CTL	
See Section 16.5.2, “Package level Enabling HDC.”		Package
Register Address: DB1H, 3505	IA32_PM_CTL1	
See Section 16.5.3, “Logical-Processor Level HDC Control.”		Thread
Register Address: DB2H, 3506	IA32_THREAD_STALL	
See Section 16.5.4.1, “IA32_THREAD_STALL.”		Thread
Register Address: DC0H, 3520	MSR_LBR_INFO_0	
Last Branch Record 0 Additional Information (R/W) One of 32 triplet of last branch record registers on the last branch record stack. This part of the stack contains flag, TSX-related and elapsed cycle information. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.9.1, “LBR Stack.”		Thread
Register Address: DC1H, 3521	MSR_LBR_INFO_1	
Last Branch Record 1 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DC2H, 3522	MSR_LBR_INFO_2	
Last Branch Record 2 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DC3H, 3523	MSR_LBR_INFO_3	
Last Branch Record 3 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DC4H, 3524	MSR_LBR_INFO_4	
Last Branch Record 4 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DC5H, 3525	MSR_LBR_INFO_5	
Last Branch Record 5 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: DC6H, 3526	MSR_LBR_INFO_6	
Last Branch Record 6 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DC7H, 3527	MSR_LBR_INFO_7	
Last Branch Record 7 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DC8H, 3528	MSR_LBR_INFO_8	
Last Branch Record 8 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DC9H, 3529	MSR_LBR_INFO_9	
Last Branch Record 9 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DCAH, 3530	MSR_LBR_INFO_10	
Last Branch Record 10 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DCBH, 3531	MSR_LBR_INFO_11	
Last Branch Record 11 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DCCH, 3532	MSR_LBR_INFO_12	
Last Branch Record 12 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DCDH, 3533	MSR_LBR_INFO_13	
Last Branch Record 13 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DCEH, 3534	MSR_LBR_INFO_14	
Last Branch Record 14 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DCFH, 3535	MSR_LBR_INFO_15	
Last Branch Record 15 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD0H, 3536	MSR_LBR_INFO_16	
Last Branch Record 16 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD1H, 3537	MSR_LBR_INFO_17	
Last Branch Record 17 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD2H, 3538	MSR_LBR_INFO_18	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 18 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD3H, 3539	MSR_LBR_INFO_19	
Last Branch Record 19 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD4H, 3540	MSR_LBR_INFO_20	
Last Branch Record 20 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD5H, 3541	MSR_LBR_INFO_21	
Last Branch Record 21 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD6H, 3542	MSR_LBR_INFO_22	
Last Branch Record 22 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD7H, 3543	MSR_LBR_INFO_23	
Last Branch Record 23 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD8H, 3544	MSR_LBR_INFO_24	
Last Branch Record 24 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DD9H, 3545	MSR_LBR_INFO_25	
Last Branch Record 25 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DDAH, 3546	MSR_LBR_INFO_26	
Last Branch Record 26 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DDBH, 3547	MSR_LBR_INFO_27	
Last Branch Record 27 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DDCH, 3548	MSR_LBR_INFO_28	
Last Branch Record 28 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DDDH, 3549	MSR_LBR_INFO_29	
Last Branch Record 29 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DDEH, 3550	MSR_LBR_INFO_30	

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 E-Core Processors, Intel® Xeon® 6 P-Core Processors, and Intel® Series 2 Core™ Ultra Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Last Branch Record 30 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread
Register Address: DDFH, 3551	MSR_LBR_INFO_31	
Last Branch Record 31 Additional Information (R/W) See description of MSR_LBR_INFO_0.		Thread

Table 2-40 lists the MSRs of uncore PMU for Intel processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_4EH, 06_5EH, 06_8EH, 06_9EH, or 06_66H.

Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and 8th generation Intel® Core™ i3 Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 394H, 916	MSR_UNC_PERF_FIXED_CTRL	
Uncore Fixed Counter Control (R/W)		Package
19:0	Reserved.	
20	Enable overflow propagation.	
21	Reserved.	
22	Enable counting.	
63:23	Reserved.	
Register Address: 395H, 917	MSR_UNC_PERF_FIXED_CTR	
Uncore Fixed Counter		Package
43:0	Current count.	
63:44	Reserved.	
Register Address: 396H, 918	MSR_UNC_CBO_CONFIG	
Uncore C-Box Configuration Information (R/O)		Package
3:0	Specifies the number of C-Box units with programmable counters (including processor cores and processor graphics).	
63:4	Reserved.	
Register Address: 3B0H, 946	MSR_UNC_ARB_PERFCTR0	
Uncore Arb Unit, Performance Counter 0		Package
Register Address: 3B1H, 947	MSR_UNC_ARB_PERFCTR1	
Uncore Arb Unit, Performance Counter 1		Package
Register Address: 3B2H, 944	MSR_UNC_ARB_PERFEVTSEL0	
Uncore Arb Unit, Counter 0 Event Select MSR		Package
Register Address: 3B3H, 945	MSR_UNC_ARB_PERFEVTSEL1	
Uncore Arb Unit, Counter 1 Event Select MSR		Package

Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and 8th generation Intel® Core™ i3 Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 700H, 1792	MSR_UNC_CBO_0_PERFEVTSEL0	
Uncore C-Box 0, Counter 0 Event Select MSR		Package
Register Address: 701H, 1793	MSR_UNC_CBO_0_PERFEVTSEL1	
Uncore C-Box 0, Counter 1 Event Select MSR		Package
Register Address: 706H, 1798	MSR_UNC_CBO_0_PERFCTR0	
Uncore C-Box 0, Performance Counter 0		Package
Register Address: 707H, 1799	MSR_UNC_CBO_0_PERFCTR1	
Uncore C-Box 0, Performance Counter 1		Package
Register Address: 710H, 1808	MSR_UNC_CBO_1_PERFEVTSEL0	
Uncore C-Box 1, Counter 0 Event Select MSR		Package
Register Address: 711H, 1809	MSR_UNC_CBO_1_PERFEVTSEL1	
Uncore C-Box 1, Counter 1 Event Select MSR		Package
Register Address: 716H, 1814	MSR_UNC_CBO_1_PERFCTR0	
Uncore C-Box 1, Performance Counter 0		Package
Register Address: 717H, 1815	MSR_UNC_CBO_1_PERFCTR1	
Uncore C-Box 1, Performance Counter 1		Package
Register Address: 720H, 1824	MSR_UNC_CBO_2_PERFEVTSEL0	
Uncore C-Box 2, Counter 0 Event Select MSR		Package
Register Address: 721H, 1825	MSR_UNC_CBO_2_PERFEVTSEL1	
Uncore C-Box 2, Counter 1 Event Select MSR		Package
Register Address: 726H, 1830	MSR_UNC_CBO_2_PERFCTR0	
Uncore C-Box 2, Performance Counter 0		Package
Register Address: 727H, 1831	MSR_UNC_CBO_2_PERFCTR1	
Uncore C-Box 2, Performance Counter 1		Package
Register Address: 730H, 1840	MSR_UNC_CBO_3_PERFEVTSEL0	
Uncore C-Box 3, Counter 0 Event Select MSR		Package
Register Address: 731H, 1841	MSR_UNC_CBO_3_PERFEVTSEL1	
Uncore C-Box 3, Counter 1 Event Select MSR		Package
Register Address: 736H, 1846	MSR_UNC_CBO_3_PERFCTR0	
Uncore C-Box 3, Performance Counter 0		Package
Register Address: 737H, 1847	MSR_UNC_CBO_3_PERFCTR1	
Uncore C-Box 3, Performance Counter 1		Package
Register Address: E01H, 3585	MSR_UNC_PERF_GLOBAL_CTRL	
Uncore PMU Global Control		Package
0	Slice 0 select.	
1	Slice 1 select.	
2	Slice 2 select.	

Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and 8th generation Intel® Core™ i3 Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
3	Slice 3 select.	
4	Slice 4select.	
18:5	Reserved.	
29	Enable all uncore counters.	
30	Enable wake on PMI.	
31	Enable Freezing counter when overflow.	
63:32	Reserved.	
Register Address: E02H, 3586	MSR_UNC_PERF_GLOBAL_STATUS	
Uncore PMU Main Status		Package
0	Fixed counter overflowed.	
1	An ARB counter overflowed.	
2	Reserved.	
3	A CBox counter overflowed (on any slice).	
63:4	Reserved.	

2.17.1 MSRs Introduced in 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

Table 2-41 lists additional MSRs for 7th generation and 8th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_8EH or 06_9EH. For an MSR listed in Table 2-41 that also appears in the model-specific tables of prior generations, Table 2-41 supersedes prior generation tables.

Table 2-41. Additional MSRs Supported by the 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 80H, 128	MSR_TRACE_HUB_STH_ACPIBAR_BASE	
NPK Address Used by AET Messages (R/W)		Package
0	Lock Bit If set, this MSR cannot be re-written anymore. Lock bit has to be set in order for the AET packets to be directed to NPK MMIO.	
17:1	Reserved.	
63:18	ACPIBAR_BASE_ADDRESS AET target address in NPK MMIO space.	
Register Address: 1F4H, 500	MSR_PRMRR_PHYS_BASE	
Processor Reserved Memory Range Register - Physical Base Control Register (R/W)		Core
2:0	MemType PRMRR BASE MemType.	
11:3	Reserved.	
45:12	Base PRMRR Base Address.	
63:46	Reserved.	
Register Address: 1F5H, 501	MSR_PRMRR_PHYS_MASK	
Processor Reserved Memory Range Register - Physical Mask Control Register (R/W)		Core
9:0	Reserved.	
10	Lock Lock bit for the PRMRR.	
11	VLD Enable bit for the PRMRR.	
45:12	Mask PRMRR MASK bits.	
63:46	Reserved.	
Register Address: 1FBH, 507	MSR_PRMRR_VALID_CONFIG	
Valid PRMRR Configurations (R/W)		Core
0	1M supported MEE size.	
4:1	Reserved.	
5	32M supported MEE size.	
6	64M supported MEE size.	
7	128M supported MEE size.	

Table 2-41. Additional MSRs Supported by the 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:8	Reserved.	
Register Address: 2F4H, 756	MSR_UNCORE_PRMRR_PHYS_BASE[1]	
(R/W) The PRMRR range is used to protect the processor reserved memory from unauthorized reads and writes. Any IO access to this range is aborted. This register controls the location of the PRMRR range by indicating its starting address. It functions in tandem with the PRMRR mask register.		Package
11:0	Reserved.	
PAWIDTH-1:12	Range Base This field corresponds to bits PAWIDTH-1:12 of the base address memory range which is allocated to PRMRR memory.	
63:PAWIDTH	Reserved.	
Register Address: 2F5H, 757	MSR_UNCORE_PRMRR_PHYS_MASK[1]	
(R/W) This register controls the size of the PRMRR range by indicating which address bits must match the PRMRR base register value.		Package
9:0	Reserved.	
10	Lock Setting this bit locks all writeable settings in this register, including itself.	
11	Range_En Indicates whether the PRMRR range is enabled and valid.	
38:12	Range_Mask This field indicates which address bits must match PRMRR base in order to qualify as an PRMRR access.	
63:39	Reserved.	
Register Address: 620H, 1568	MSR_RING_RATIO_LIMIT	
Ring Ratio Limit (R/W) This register provides Min/Max Ratio Limits for the LLC and Ring.		Package
6:0	MAX_Ratio This field is used to limit the max ratio of the LLC/Ring.	
7	Reserved.	
14:8	MIN_Ratio Writing to this field controls the minimum possible ratio of the LLC/Ring.	
63:15	Reserved.	

NOTES:

1. This MSR is specific to 7th generation and 8th generation Intel® Core™ processors.

2.17.2 MSRs Specific to 8th Generation Intel® Core™ i3 Processors

Table 2-42 lists additional MSRs for 8th generation Intel Core i3 processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_66H. For an MSR listed in Table 2-42 that also appears in the model-specific tables of prior generations, Table 2-42 supersedes prior generation tables.

Table 2-42. Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64 Processor (R/W) See Table 2-2.		Thread
0	Lock (R/WL)	
1	Enable VMX Inside SMX Operation (R/WL)	
2	Enable VMX Outside SMX Operation (R/WL)	
14:8	SENTER Local Functions Enables (R/WL)	
15	SENTER Global Functions Enable (R/WL)	
17	SGX Launch Control Enable (R/WL) This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Available only if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.	
18	SGX Global Functions Enable (R/WL)	
63:21	Reserved.	
Register Address: 350H, 848	MSR_BR_DETECT_CTRL	
Branch Monitoring Global Control (R/W)		
0	EnMonitoring Global enable for branch monitoring.	
1	EnExcept Enable branch monitoring event signaling on threshold trip. The branch monitoring event handler is signaled via the existing PMI signaling mechanism as programmed from the corresponding local APIC LVT entry.	
2	EnLBRFrz Enable LBR freeze on threshold trip. This will cause the LBR frozen bit 58 to be set in IA32_PERF_GLOBAL_STATUS when a triggering condition occurs and this bit is enabled.	
3	DisableInGuest When set to '1', branch monitoring, event triggering and LBR freeze actions are disabled when operating at VMX non-root operation.	
7:4	Reserved.	
17:8	WindowSize Window size defined by WindowCntSel. Values 0 – 1023 are supported. Once the Window counter reaches the WindowSize count both the Window Counter and all Branch Monitoring Counters are cleared.	
23:18	Reserved.	

Table 2-42. Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
25:24	WindowCntSel Window event count select: '00 = Instructions retired. '01 = Branch instructions retired '10 = Return instructions retired. '11 = Indirect branch instructions retired.	
26	CntAndMode When set to '1', the overall branch monitoring event triggering condition is true only if all enabled counters' threshold conditions are true. When '0', the threshold tripping condition is true if any enabled counters' threshold is true.	
63:27	Reserved.	
Register Address: 351H, 849	MSR_BR_DETECT_STATUS	
Branch Monitoring Global Status (R/W)		
0	Branch Monitoring Event Signaled When set to '1', Branch Monitoring event signaling is blocked until this bit is cleared by software.	
1	LBRsValid This status bit is set to '1' if the LBR state is considered valid for sampling by branch monitoring software.	
7:2	Reserved.	
8	CntrHit0 Branch monitoring counter #0 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.	
9	CntrHit1 Branch monitoring counter #1 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.	
15:10	Reserved. Reserved for additional branch monitoring counters threshold hit status.	
25:16	CountWindow The current value of the window counter. The count value is frozen on a valid branch monitoring triggering condition. This is a 10-bit unsigned value.	
31:26	Reserved. Reserved for future extension of CountWindow.	

Table 2-42. Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
39:32	Count0 The current value of counter 0 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit0 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).	
47:40	Count1 The current value of counter 1 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit1 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).	
63:48	Reserved.	
Register Address: 354H—355H, 852—853	MSR_BR_DETECT_COUNTER_CONFIG_i	
Branch Monitoring Detect Counter Configuration (R/W)		
0	CntrEn Enable counter.	
7:1	CntrEvSel Event select (other values #GP) '0000000 = RETs. '0000001 = RET-CALL bias. '0000010 = RET mispredicts. '0000011 = Branch (all) mispredicts. '0000100 = Indirect branch mispredicts. '0000101 = Far branch instructions.	
14:8	CntrThreshold Threshold (an unsigned value of 0 to 127 supported). The value 0 of counter threshold will result in event signaled after every instruction. #GP if threshold is < 2.	
15	MispredEventCnt Mispredict events counting behavior: '0 = Mispredict events are counted in a window. '1 = Mispredict events are counted based on a consecutive occurrence. CntrThreshold is treated as # of consecutive mispredicts. This control bit only applies to events specified by CntrEvSel that involve a prediction (0000010, 0000011, 0000100). Setting this bit for other events is ignored.	
63:16	Reserved.	

Table 2-42. Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3F8H, 1016	MSR_PKG_C3_RESIDENCY	
Package C3 Residency Counter (R/O)		Package
63:0	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.	
Register Address: 620H, 1568	MSR_RING_RATIO_LIMIT	
Ring Ratio Limit (R/W) This register provides Min/Max Ratio Limits for the LLC and Ring.		Package
6:0	MAX_Ratio This field is used to limit the max ratio of the LLC/Ring.	
7	Reserved.	
14:8	MIN_Ratio Writing to this field controls the minimum possible ratio of the LLC/Ring.	
63:15	Reserved.	
Register Address: 660H, 1632	MSR_CORE_C1_RESIDENCY	
Core C1 Residency Counter (R/O)		Core
63:0	Value since last reset for the Core C1 residency. Counter rate is the Max Non-Turbo frequency (same as TSC). This counter counts in case both of the core's threads are in an idle state and at least one of the core's thread residency is in a C1 state or in one of its sub states. The counter is updated only after a core C state exit. Note: Always reads 0 if core C1 is unsupported. A value of zero indicates that this processor does not support core C1 or never entered core C1 level state.	
Register Address: 662H, 1634	MSR_CORE_C3_RESIDENCY	
Core C3 Residency Counter (R/O)		Core
63:0	Will always return 0.	

Table 2-43 lists the MSRs of uncore PMU for Intel processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_66H.

Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 394H, 916	MSR_UNC_PERF_FIXED_CTRL	
Uncore Fixed Counter Control (R/W)		Package
19:0	Reserved.	
20	Enable overflow propagation.	
21	Reserved	
22	Enable counting.	
63:23	Reserved.	
Register Address: 395H, 917	MSR_UNC_PERF_FIXED_CTR	

Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore Fixed Counter		Package
47:0	Current count.	
63:48	Reserved.	
Register Address: 396H, 918	MSR_UNC_CBO_CONFIG	
Uncore C-Box Configuration Information (R/O)		Package
3:0	Report the number of C-Box units with performance counters, including processor cores and processor graphics.	
63:4	Reserved.	
Register Address: 3B0H, 946	MSR_UNC_ARB_PERFCTR0	
Uncore Arb Unit, Performance Counter 0		Package
Register Address: 3B1H, 947	MSR_UNC_ARB_PERFCTR1	
Uncore Arb Unit, Performance Counter 1		Package
Register Address: 3B2H, 944	MSR_UNC_ARB_PERFEVTSEL0	
Uncore Arb Unit, Counter 0 Event Select MSR		Package
Register Address: 3B3H, 945	MSR_UNC_ARB_PERFEVTSEL1	
Uncore Arb unit, Counter 1 Event Select MSR		Package
Register Address: 700H, 1792	MSR_UNC_CBO_0_PERFEVTSEL0	
Uncore C-Box 0, Counter 0 Event Select MSR		Package
Register Address: 701H, 1793	MSR_UNC_CBO_0_PERFEVTSEL1	
Uncore C-Box 0, Counter 1 Event Select MSR		Package
Register Address: 702H, 1794	MSR_UNC_CBO_0_PERFCTR0	
Uncore C-Box 0, Performance Counter 0		Package
Register Address: 703H, 1795	MSR_UNC_CBO_0_PERFCTR1	
Uncore C-Box 0, Performance Counter 1		Package
Register Address: 708H, 1800	MSR_UNC_CBO_1_PERFEVTSEL0	
Uncore C-Box 1, Counter 0 Event Select MSR		Package
Register Address: 709H, 1801	MSR_UNC_CBO_1_PERFEVTSEL1	
Uncore C-Box 1, Counter 1 Event Select MSR		Package
Register Address: 70AH, 1802	MSR_UNC_CBO_1_PERFCTR0	
Uncore C-Box 1, Performance Counter 0		Package
Register Address: 70BH, 1803	MSR_UNC_CBO_1_PERFCTR1	
Uncore C-Box 1, Performance Counter 1		Package
Register Address: 710H, 1808	MSR_UNC_CBO_2_PERFEVTSEL0	
Uncore C-Box 2, Counter 0 Event Select MSR		Package
Register Address: 711H, 1809	MSR_UNC_CBO_2_PERFEVTSEL1	
Uncore C-Box 2, Counter 1 Event Select MSR		Package
Register Address: 712H, 1810	MSR_UNC_CBO_2_PERFCTR0	
Uncore C-Box 2, Performance Counter 0		Package

Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 713H, 1811	MSR_UNC_CBO_2_PERFCTR1	
Uncore C-Box 2, Performance Counter 1		Package
Register Address: 718H, 1816	MSR_UNC_CBO_3_PERFEVTSEL0	
Uncore C-Box 3, Counter 0 Event Select MSR		Package
Register Address: 719H, 1817	MSR_UNC_CBO_3_PERFEVTSEL1	
Uncore C-Box 3, Counter 1 Event Select MSR		Package
Register Address: 71AH, 1818	MSR_UNC_CBO_3_PERFCTR0	
Uncore C-Box 3, Performance Counter 0		Package
Register Address: 71BH, 1819	MSR_UNC_CBO_3_PERFCTR1	
Uncore C-Box 3, Performance Counter 1		Package
Register Address: 720H, 1824	MSR_UNC_CBO_4_PERFEVTSEL0	
Uncore C-Box 4, Counter 0 Event Select MSR		Package
Register Address: 721H, 1825	MSR_UNC_CBO_4_PERFEVTSEL1	
Uncore C-Box 4, Counter 1 Event Select MSR		Package
Register Address: 722H, 1826	MSR_UNC_CBO_4_PERFCTR0	
Uncore C-Box 4, Performance Counter 0		Package
Register Address: 723H, 1827	MSR_UNC_CBO_4_PERFCTR1	
Uncore C-Box 4, Performance Counter 1		Package
Register Address: 728H, 1832	MSR_UNC_CBO_5_PERFEVTSEL0	
Uncore C-Box 5, Counter 0 Event Select MSR		Package
Register Address: 729H, 1833	MSR_UNC_CBO_5_PERFEVTSEL1	
Uncore C-Box 5, Counter 1 Event Select MSR		Package
Register Address: 72AH, 1834	MSR_UNC_CBO_5_PERFCTR0	
Uncore C-Box 5, Performance Counter 0		Package
Register Address: 72BH, 1835	MSR_UNC_CBO_5_PERFCTR1	
Uncore C-Box 5, Performance Counter 1		Package
Register Address: 730H, 1840	MSR_UNC_CBO_6_PERFEVTSEL0	
Uncore C-Box 6, Counter 0 Event Select MSR		Package
Register Address: 731H, 1841	MSR_UNC_CBO_6_PERFEVTSEL1	
Uncore C-Box 6, Counter 1 Event Select MSR		Package
Register Address: 732H, 1842	MSR_UNC_CBO_6_PERFCTR0	
Uncore C-Box 6, Performance Counter 0		Package
Register Address: 733H, 1843	MSR_UNC_CBO_6_PERFCTR1	
Uncore C-Box 6, Performance Counter 1		Package
Register Address: 738H, 1848	MSR_UNC_CBO_7_PERFEVTSEL0	
Uncore C-Box 7, Counter 0 Event Select MSR		Package
Register Address: 739H, 1849	MSR_UNC_CBO_7_PERFEVTSEL1	

Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 7, Counter 1 Event Select MSR		Package
Register Address: 73AH, 1850	MSR_UNC_CBO_7_PERFCTR0	
Uncore C-Box 7, Performance Counter 0		Package
Register Address: 73BH, 1851	MSR_UNC_CBO_7_PERFCTR1	
Uncore C-Box 7, Performance Counter 1		Package
Register Address: E01H, 3585	MSR_UNC_PERF_GLOBAL_CTRL	
Uncore PMU Global Control		Package
0	Slice 0 select.	
1	Slice 1 select.	
2	Slice 2 select.	
3	Slice 3 select.	
4	Slice 4select.	
18:5	Reserved.	
29	Enable all uncore counters.	
30	Enable wake on PMI.	
31	Enable Freezing counter when overflow.	
63:32	Reserved.	
Register Address: E02H, 3586	MSR_UNC_PERF_GLOBAL_STATUS	
Uncore PMU Main Status		Package
0	Fixed counter overflowed.	
1	An ARB counter overflowed.	
2	Reserved.	
3	A CBox counter overflowed (on any slice).	
63:4	Reserved.	

2.17.3 MSRs Introduced in 10th Generation Intel® Core™ Processors

Table 2-44 lists additional MSRs for 10th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_7DH or 06_7EH. For an MSR listed in Table 2-44 that also appears in the model-specific tables of prior generations, Table 2-44 supersedes prior generation tables.

Table 2-44. MSRs Supported by the 10th Generation Intel® Core™ Processors (Ice Lake Microarchitecture)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 33H, 51	MSR_MEMORY_CTRL	
Memory Control Register		Core
28:0	Reserved.	

Table 2-44. MSRs Supported by the 10th Generation Intel® Core™ Processors (Ice Lake Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
29	SPLIT_LOCK_DISABLE If set to 1, a split lock will cause an #AC(0) exception. See Section 10.1.2.3, "Features to Disable Bus Locks."	
30	Reserved.	
31	Reserved.	
Register Address: 48H, 72	IA32_SPEC_CTRL	
See Table 2-2.		Core
Register Address: 49H, 73	IA32_PREDICT_CMD	
See Table 2-2.		Thread
Register Address: 8CH, 140	IA32_SGXLEPUBKEYHASH0	
See Table 2-2.		Thread
Register Address: 8DH, 141	IA32_SGXLEPUBKEYHASH1	
See Table 2-2.		Thread
Register Address: 8EH, 142	IA32_SGXLEPUBKEYHASH2	
See Table 2-2.		Thread
Register Address: 8FH, 143	IA32_SGXLEPUBKEYHASH3	
See Table 2-2.		Thread
Register Address: A0H, 160	MSR_BIOS_MCU_ERRORCODE	
BIOS MCU ERRORCODE (R/O) This MSR indicates if WRMSR 0x79 failed to configure PRM memory and gives a hint to debug BIOS.		Package
15:0	Error Codes (R/O)	Package
30:16	Reserved.	
31	MCU Partial Success (R/O) When set to 1, WRMSR 0x79 skipped part of the functionality during BIOS.	Thread
Register Address: A5H, 165	MSR_FIT_BIOS_ERROR	
FIT BIOS ERROR (R/W) Report error codes for debug in case the processor failed to parse the Firmware Table in BIOS. Can also be used to log BIOS information.		Thread
7:0	Error Codes (R/W) Error codes for debug.	
15:8	Entry Type (R/W) Failed FIT entry type.	
16	FIT MCU Entry (R/W) FIT contains MCU entry.	
62:17	Reserved.	
63	LOCK (R/W) When set to 1, writes to this MSR will be skipped.	
Register Address: 10BH, 267	IA32_FLUSH_CMD	

Table 2-44. MSRs Supported by the 10th Generation Intel® Core™ Processors (Ice Lake Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2.		Thread
Register Address: 151H, 337	MSR_BIOS_DONE	
BIOS Done (R/WO)		Thread
0	BIOS Done Indication (R/WO) Set by BIOS when it finishes programming the processor and wants to lock the memory configuration from changes by software that is running on this thread. Writes to the bit will be ignored if EAX[0] is 0.	Thread
1	Package BIOS Done Indication (R/O) When set to 1, all threads in the package have bit 0 of this MSR set.	Package
31:2	Reserved.	
Register Address: 1F1H, 497	MSR_CRASHLOG_CONTROL	
Write Data to a Crash Log Configuration		Thread
0	CDDIS: CrashDump_Disable If set, indicates that Crash Dump is disabled.	
63:1	Reserved.	
Register Address: 2A0H, 672	MSR_PRMRR_BASE_0	
Processor Reserved Memory Range Register - Physical Base Control Register (R/W)		Core
2:0	MEMTYPE: PRMRR BASE Memory Type.	
3	CONFIGURED: PRMRR BASE Configured.	
11:4	Reserved.	
51:12	BASE: PRMRR Base Address.	
63:52	Reserved.	
Register Address: 30CH, 780	IA32_FIXED_CTR3	
Fixed-Function Performance Counter Register 3 (R/W) Bit definitions are the same as found in IA32_FIXED_CTR0, offset 309H. See Table 2-2.		Thread
Register Address: 329H, 809	MSR_PERF_METRICS	
Performance Metrics (R/W) Reports metrics directly. Software can check (and/or expose to its guests) the availability of PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15).		Thread
7:0	Retiring. Percent of utilized slots by uops that eventually retire (commit).	
15:8	Bad Speculation. Percent of wasted slots due to incorrect speculation, covering utilized by uops that do not retire, or recovery bubbles (unutilized slots).	
23:16	Frontend Bound. Percent of unutilized slots where front-end did not deliver a uop while back-end is ready.	
31:24	Backend Bound. Percent of unutilized slots where a uop was not delivered to back-end due to lack of back-end resources.	
63:32	Reserved.	
Register Address: 3F2H, 1010	MSR_PEBS_DATA_CFG	

Table 2-44. MSRs Supported by the 10th Generation Intel® Core™ Processors (Ice Lake Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
PEBS Data Configuration (R/W) Provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.		Thread
0	Memory Info. Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.	
1	GPRs. Setting this bit will capture the contents of the General Purpose registers in the PEBS record.	
2	XMMs. Setting this bit will capture the contents of the XMM registers in the PEBS record.	
3	LBRs. Setting this bit will capture LBR TO, FROM, and INFO in the PEBS record.	
23:4	Reserved.	
31:24	LBR Entries. Set the field to the desired number of entries - 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, software can use LBR_entries that is multiple of 3.	
Register Address: 541H, 1345	MSR_CORE_UARCH_CTL	
Core Microarchitecture Control MSR (R/W)		Core
0	L1 Scrubbing Enable When set to 1, enable L1 scrubbing.	
31:1	Reserved.	
Register Address: 657H, 1623	MSR_FAST_UNCORE_MSRS_CTL	
Fast WRMSR/RDMSR Control MSR (R/W)		Thread
3:0	FAST_ACCESS_ENABLE: Bit 0: When set to '1', provides a hint for the hardware to enable fast access mode for the IA32_HWP_REQUEST MSR. This bit is sticky and is cleaned by the hardware only during reset time. This bit is valid only if FAST_UNCORE_MSRS_CAPABILITY[0] is set. Setting this bit will cause CPUID[6].EAX[18] to be set.	
31:4	Reserved.	
Register Address: 65EH, 1630	MSR_FAST_UNCORE_MSRS_STATUS	
Indication of Uncore MSRs, Post Write Activates		Thread

Table 2-44. MSRs Supported by the 10th Generation Intel® Core™ Processors (Ice Lake Microarchitecture) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
0	Indicates whether the CPU is still in the middle of writing IA32_HWP_REQUEST MSR, even after the WRMSR instruction has retired. A value of 1 indicates the last write of IA32_HWP_REQUEST is still ongoing. A value of 0 indicates the last write of IA32_HWP_REQUEST is visible outside the logical processor. Software can use the status of this bit to avoid overwriting IA32_HWP_REQUEST.	
31:1	Reserved.	
Register Address: 65FH, 1631	MSR_FAST_UNCORE_MSRS_CAPABILITY	
Fast WRMSR/RDMSR Enumeration MSR (R/O)		Thread
3:0	MSRS_CAPABILITY: Bit 0: If set to '1', hardware supports the fast access mode for the IA32_HWP_REQUEST MSR.	
31:4	Reserved.	
Register Address: 772H, 1906	IA32_HWP_REQUEST_PKG	
See Table 2-2.		Package
Register Address: 775H, 1909	IA32_PECI_HWP_REQUEST_INFO	
See Table 2-2.		Package
Register Address: 777H, 1911	IA32_HWP_STATUS	
See Table 2-2.		Thread

2.17.4 MSRs Introduced in the 11th Generation Intel® Core™ Processors based on Tiger Lake Microarchitecture

Table 2-45 lists additional MSRs for 11th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_8CH or 06_8DH. The MSRs listed in Table 2-44 are also supported by these processors. For an MSR listed in Table 2-45 that also appears in the model-specific tables of prior generations, Table 2-45 supersedes prior generation tables.

Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: A0H, 160	MSR_BIOS_MCU_ERRORCODE	
BIOS MCU ERRORCODE (R/O)		Package
15:0	Error Codes	
31:16	Reserved.	
Register Address: A7H, 167	MSR_BIOS_DEBUG	
BIOS DEBUG (R/O) This MSR indicates if WRMSR 79H failed to configure PRM memory and gives a hint to debug BIOS.		Thread
30:0	Reserved.	

Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31	MCU Partial Success When set to 1, WRMSR 79H skipped part of the functionality during BIOS.	
63:32	Reserved.	
Register Address: CFH, 207	IA32_CORE_CAPABILITIES	
IA32 Core Capabilities Register (R/O) If CPUID.(EAX=07H, ECX=0):EDX[30] = 1. This MSR provides an architectural enumeration function for model-specific behavior.		Package
1:0	Reserved.	
2	FUSA_SUPPORTED	
3	RSM_IN_CPL0_ONLY When set to 1, the RSM instruction is only allowed in CPL0 (#GP triggered in any CPL != 0). When set to 0, then any CPL may execute the RSM instruction.	
4	Reserved.	
5	SPLIT_LOCK_DISABLE_SUPPORTED When read as 1, software can set bit 29 of MSR_MEMORY_CTRL (MSR address 33H).	
31:6	Reserved.	
Register Address: 492H, 1170	IA32_VMX_PROCBASED_CTLS3	
IA32_VMX_PROCBASED_CTLS3 This MSR enumerates the allowed 1-settings of the third set of processor-based controls. Specifically, VM entry allows bit X of the tertiary processor-based VM-execution controls to be 1 if and only if bit X of the MSR is set to 1. If bit X of the MSR is cleared to 0, VM entry fails if control X and the “activate tertiary controls” primary processor-based VM-execution control are both 1.		Core
0	LOADIWKEY This control determines whether executions of LOADIWKEY cause VM exits.	
63:1	Reserved.	
Register Address: 601H, 1537	MSR_VR_CURRENT_CONFIG	
Power Limit 4 (PL4) Package-level maximum power limit (in Watts). It is a proactive, instantaneous limit.		Package
12:0	PL4 Value PL4 value in 0.125 A increments. This field is locked by VR_CURRENT_CONFIG[LOCK]. When the LOCK bit is set to 1b, this field becomes Read Only.	
30:13	Reserved.	
31	Lock Indication (LOCK) This bit will lock the CURRENT_LIMIT settings in this register and will also lock this setting. This means that once set to 1b, the CURRENT_LIMIT setting and this bit become Read Only until the next Warm Reset.	

Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
62:32	Not in use.	
63	Reserved.	
Register Address: 6A0H, 1696	IA32_U_CET	
Configure User Mode CET (R/W) See Table 2-2.		
Register Address: 6A2H, 1698	IA32_S_CET	
Configure Supervisor Mode CET (R/W) See Table 2-2.		
Register Address: 6A4H, 1700	IA32_PL0_SSP	
Linear address to be loaded into SSP on transition to privilege level 0. (R/W) See Table 2-2.		
Register Address: 6A5H, 1701	IA32_PL1_SSP	
Linear address to be loaded into SSP on transition to privilege level 1. (R/W) See Table 2-2.		
Register Address: 6A6H, 1702	IA32_PL2_SSP	
Linear address to be loaded into SSP on transition to privilege level 2. (R/W) See Table 2-2.		
Register Address: 6A7H, 1703	IA32_PL3_SSP	
Linear address to be loaded into SSP on transition to privilege level 3. (R/W) See Table 2-2.		
Register Address: 6A8H, 1704	IA32_INTERRUPT_SSP_TABLE_ADDR	
Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W) See Table 2-2.		
Register Address: 981H, 2433	IA32_TME_CAPABILITY	
See Table 2-2.		
Register Address: 982H, 2434	IA32_TME_ACTIVATE	
See Table 2-2.		
Register Address: 983H, 2435	IA32_TME_EXCLUDE_MASK	
See Table 2-2.		
Register Address: 984H, 2436	IA32_TME_EXCLUDE_BASE	
See Table 2-2.		
Register Address: 990H, 2448	IA32_COPY_STATUS[1]	
See Table 2-2.		Thread
Register Address: 991H, 2449	IA32_IWKEYBACKUP_STATUS[1]	
See Table 2-2.		Platform
Register Address: C82H, 3202	IA32_L2_QOS_CFG	

Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32_CR_L2_QOS_CFG This MSR provides software an enumeration of the parameters that L2 QoS (Intel RDT) support in any particular implementation.		Core
0	CDP_ENABLE When set to 1, it will enable the code and data prioritization for the L2 CAT/Intel RDT feature. When set to 0, code and data prioritization is disabled for L2 CAT/Intel RDT. See Chapter 19, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features,” for further details on CDP.	
31:1	Reserved.	
Register Address: D10H—D17H, 3220—3351	IA32_L2_QOS_MASK_[0-7]	
IA32_CR_L2_QOS_MASK_[0-7] Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 19, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features.”		Package
19:0	WAYS_MASK Setting a 1 in this bit X allows threads with CLOS <n> (where N is [0-7]) to allocate to way X in the MLC. Ones are only allowed to be written to ways that physically exist in the MLC (CPUID.4.2:EBX[31:22] will indicate this). Writing a 1 to a value beyond the highest way or a non-contiguous set of 1s will cause a #GP on the WRMSR to this MSR.	
31:20	Reserved.	
Register Address: D91H, 3473	IA32_COPY_LOCAL_TO_PLATFORM[1]	
See Table 2-2.		Thread
Register Address: D92H, 3474	IA32_COPY_PLATFORM_TO_LOCAL[1]	
See Table 2-2.		Thread

NOTES:

1. Further details on Key Locker and usage of this MSR can be found here:
   https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

2.17.5 MSRs Introduced in the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture

Table 2-46 lists additional MSRs for 12th and 13th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_97H, 06_9AH, 06_BAH, 06_B7H, or 06_BFH. Table 2-47 lists the MSRs unique to the processor P-core. Table 2-48 lists the MSRs unique to the processor E-core.

The MSRs listed in Table 2-44[1] and Table 2-45 are also supported by these processors. For an MSR listed in Table 2-46, Table 2-47, or Table 2-48 that also appears in the model-specific tables of prior generations, Table 2-46, Table 2-47, and Table 2-48 supersede prior generation tables.

1. MSRs at the following addresses are not supported in the 12th and 13th generation Intel Core processor E-core: 30CH, 329H, 541H, and 657H. The MSR at address 657H is not supported in the 12th and 13th generation Intel Core processor P-core.

Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 33H, 51	MSR_MEMORY_CTRL	
Memory Control Register		Core
26:0	Reserved.	
27	UC_STORE_THROTTLE If set to 1, when enabled, the processor will only allow one in-progress UC store at a time.	
28	UC_LOCK_DISABLE If set to 1, a UC lock will cause a #GP(0) exception. See Section 10.1.2.3, “Features to Disable Bus Locks.”	
29	SPLIT_LOCK_DISABLE If set to 1, a split lock will cause an #AC(0) exception. See Section 10.1.2.3, “Features to Disable Bus Locks.”	
30	Reserved.	
31	Reserved.	
Register Address: BCH, 188	IA32_MISC_PACKAGE_CTLS	
Power Filtering Control (R/W) IA32_ARCH_CAPABILITIES[bit 10] enumerates support for this MSR. See Table 2-2.		Package
Register Address: C7H, 199	IA32_PMC6	
General Performance Counter 6 (R/W) See Table 2-2.		Core
Register Address: C8H, 200	IA32_PMC7	
General Performance Counter 7 (R/W) See Table 2-2.		Core
Register Address: CFH, 207	IA32_CORE_CAPABILITIES	
IA32 Core Capabilities Register (R/O) If CPUID.(EAX=07H, ECX=0):EDX[30] = 1. This MSR provides an architectural enumeration function for model-specific behavior.		Package
0	STLB_QOS_SUPPORTED When set to 1, the STLB QoS feature is supported and the STLB QoS MSRs (1A8FH -1A97H) are accessible. When set to 0, access to these MSRs will #GP.	
1	Reserved.	
2	FUSA_SUPPORTED	
3	RSM_IN_CPL0_ONLY When set to 1, the RSM instruction is only allowed in CPL0 (#GP triggered in any CPL != 0). When set to 0, then any CPL may execute the RSM instruction.	

Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
4	UC_LOCK_DISABLE_SUPPORTED When read as 1, software can set bit 28 of MSR_MEMORY_CTRL (MSR address 33H).	
5	SPLIT_LOCK_DISABLE_SUPPORTED When read as 1, software can set bit 29 of MSR_MEMORY_CTRL.	
6	SNOOP_FILTER_QOS_SUPPORTED When set to 1, the Snoop Filter Qos Mask MSRs are supported. When set to 0, access to these MSRs will #GP.	
7	UC_STORE_THROTTLING_SUPPORTED When set 1, UC Store throttle capability exist through MSR_MEMORY_CTRL (33H) bit 27.	
31:8	Reserved.	
Register Address: E1H, 225	IA32_UMWAIT_CONTROL	
UMWAIT Control (R/W) See Table 2-2.		
Register Address: 10AH, 266	IA32_ARCH_CAPABILITIES	
Enumeration of Architectural Features (R/O) See Table 2-2.		
Register Address: 18CH, 396	IA32_PERFEVTSEL6	
See Table 2-20.		Core
Register Address: 18DH, 397	IA32_PERFEVTSEL7	
See Table 2-20.		Core
Register Address: 195H, 405	IA32_OVERCLOCKING_STATUS	
Overclocking Status (R/O) IA32_ARCH_CAPABILITIES[bit 23] enumerates support for this MSR. See Table 2-2.		Package
Register Address: 1ADH, 429	MSR_PRIMARY_TURBO_RATIO_LIMIT	
Primary Maximum Turbo Ratio Limit (R/W) Software can configure these limits when MSR_PLATFORM_INFO[28] = 1. Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.		Package
7:0	MAX_TURBO_GROUP_0: Maximum turbo ratio limit with 1 core active.	
15:8	MAX_TURBO_GROUP_1: Maximum turbo ratio limit with 2 cores active.	
23:16	MAX_TURBO_GROUP_2: Maximum turbo ratio limit with 3 cores active.	
31:24	MAX_TURBO_GROUP_3: Maximum turbo ratio limit with 4 cores active.	
39:32	MAX_TURBO_GROUP_4: Maximum turbo ratio limit with 5 cores active.	

Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
47:40	MAX_TURBO_GROUP_5: Maximum turbo ratio limit with 6 cores active.	
55:48	MAX_TURBO_GROUP_6: Maximum turbo ratio limit with 7 cores active.	
63:56	MAX_TURBO_GROUP_7: Maximum turbo ratio limit with 8 cores active.	
Register Address: 493H, 1171	IA32_VMX_EXIT_CTLS2	
See Table 2-2.		
Register Address: 4C7H, 1223	IA32_A_PMC6	
Full Width Writable IA32_PMC6 Alias (R/W) See Table 2-2.		
Register Address: 4C8H, 1224	IA32_A_PMC7	
Full Width Writable IA32_PMC7 Alias (R/W) See Table 2-2.		
Register Address: 650H, 1616	MSR_SECONDARY_TURBO_RATIO_LIMIT	
Secondary Maximum Turbo Ratio Limit (R/W) Software can configure these limits when MSR_PLATFORM_INFO[28] = 1. Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.		Package
7:0	MAX_TURBO_GROUP_0: Maximum turbo ratio limit with 1 core active.	
15:8	MAX_TURBO_GROUP_1: Maximum turbo ratio limit with 2 cores active.	
23:16	MAX_TURBO_GROUP_2: Maximum turbo ratio limit with 3 cores active.	
31:24	MAX_TURBO_GROUP_3: Maximum turbo ratio limit with 4 cores active.	
39:32	MAX_TURBO_GROUP_4: Maximum turbo ratio limit with 5 cores active.	
47:40	MAX_TURBO_GROUP_5: Maximum turbo ratio limit with 6 cores active.	
55:48	MAX_TURBO_GROUP_6: Maximum turbo ratio limit with 7 cores active.	
63:56	MAX_TURBO_GROUP_7: Maximum turbo ratio limit with 8 cores active.	
Register Address: 664H, 1636	MSR_MC6_RESIDENCY_COUNTER	
Module C6 Residency Counter (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Module

Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:0	Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.	
Register Address: 6E1H, 1761	IA32_PKRS	
Specifies the PK permissions associated with each protection domain for supervisor pages (R/W) See Table 2-2.		
Register Address: 776H, 1910	IA32_HWP_CTL	
See Table 2-2.		
Register Address: 981H, 2433	IA32_TME_CAPABILITY	
Memory Encryption Capability MSR See Table 2-2.		
Register Address: 1200H—121FH, 4608—4639	IA32_LBR_x_INFO	
Last Branch Record Entry X Info Register (R/W) See Table 2-2.		
Register Address: 14CEH, 5326	IA32_LBR_CTL	
Last Branch Record Enabling and Configuration Register (R/W) See Table 2-2.		
Register Address: 14CFH, 5327	IA32_LBR_DEPTH	
Last Branch Record Maximum Stack Depth Register (R/W) See Table 2-2.		
Register Address: 1500H—151FH, 5376—5407	IA32_LBR_x_FROM_IP	
Last Branch Record Entry X Source IP Register (R/W) See Table 2-2.		
Register Address: 1600H—161FH, 5632—5663	IA32_LBR_x_TO_IP	
Last Branch Record Entry X Destination IP Register (R/W) See Table 2-2.		
Register Address: 17D2H, 6098	IA32_THREAD_FEEDBACK_CHAR	
Thread Feedback Characteristics (R/O) See Table 2-2.		
Register Address: 17D4H, 6100	IA32_HW_FEEDBACK_THREAD_CONFIG	
Hardware Feedback Thread Configuration (R/W) See Table 2-2.		
Register Address: 17DAH, 6106	IA32_HRESET_ENABLE	
History Reset Enable (R/W) See Table 2-2.		

The MSRs listed in Table 2-47 are unique to the 12th and 13th generation Intel Core processor P-core. These MSRs are not supported on the processor E-core.

Table 2-47. MSRs Supported by 12th and 13th Generation Intel® Core™ Processor P-core

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1A4H, 420	MSR_PREFETCH_CONTROL	
Prefetch Disable Bits (R/W)		
0	L2_HARDWARE_PREFETCHER_DISABLE If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.	
1	L2_ADJACENT_CACHE_LINE_PREFETCHER_DISABLE If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).	
2	DCU_HARDWARE_PREFETCHER_DISABLE If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.	
3	DCU_IP_PREFETCHER_DISABLE If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.	
4	Reserved.	
5	AMP_PREFETCH_DISABLE If 1, disables the L2 Adaptive Multipath Probability (AMP) prefetcher.	
63:6	Reserved.	
Register Address: 3F7H, 1015	MSR_PEBS_FRONTEND	
FrontEnd Precise Event Condition Select (R/W) See Table 2-39.		Thread
Register Address: 540H, 1344	MSR_THREAD_UARCH_CTL	
Thread Microarchitectural Control (R/W)		Thread
0	WB_MEM_STRM_LD_DISABLE Disable streaming behavior for MOVNTDQA loads to WB memory type. If set, these accesses will be treated like regular cacheable loads (Data will be cached).	
63:1	Reserved.	
Register Address: 541H, 1345	MSR_CORE_UARCH_CTL	
Core Microarchitecture Control MSR (R/W) See Table 2-44.		Core
Register Address: D10H—D17H, 3220—3351	IA32_L2_QOS_MASK_[0-7]	
IA32_CR_L2_QOS_MASK_[0-7] If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] $\geq$ 0. Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 19, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features."		Core

Table 2-47. MSRs Supported by 12th and 13th Generation Intel® Core™ Processor P-core

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
19:0	WAYS_MASK Setting a 1 in this bit X allows threads with CLOS <n> (where N is [0-7]) to allocate to way X in the MLC. Ones are only allowed to be written to ways that physically exist in the MLC (CPUID.4.2:EBX[31:22] will indicate this). Writing a 1 to a value beyond the highest way or a non-contiguous set of 1s will cause a #GP on the WRMSR to this MSR.	
31:20	Reserved.	

The MSRs listed in Table 2-48 are unique to the 12th and 13th generation Intel Core processor E-core. These MSRs are not supported on the processor P-core.

Table 2-48. MSRs Supported by 12th and 13th Generation Intel® Core™ Processor E-core

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: D10H—D1FH, 3220—3359	IA32_L2_QOS_MASK_[0-15]	
IA32_CR_L2_QOS_MASK_[0-15] If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] ≥ 0. Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 19, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features."		Module
19:0	WAYS_MASK Setting a 1 in this bit X allows threads with CLOS <n> (where N is [0-7]) to allocate to way X in the MLC. Ones are only allowed to be written to ways that physically exist in the MLC (CPUID.4.2:EBX[31:22] will indicate this). Writing a 1 to a value beyond the highest way or a non-contiguous set of 1s will cause a #GP on the WRMSR to this MSR.	
31:20	Reserved.	
Register Address: 1309H—130BH, 4873—4875	MSR_RELOAD_FIXED_CTRx	
Reload value for IA32_FIXED_CTRx (R/W)		
47:0	Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and FIXED_CTRx is overflowed.	
63:48	Reserved.	
Register Address: 14C1H—14C6H, 5313 —5318	MSR_RELOAD_PMCx	
Reload value for IA32_PMCx (R/W)		Core
47:0	Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and PMCx is overflowed.	
63:48	Reserved.	

Table 2-49 lists the MSRs of uncore PMU for Intel processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_97H, 06_9AH, 06_BAH, 06_B7H, or 06_BFH.

Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 396H, 918	MSR_UNC_CBO_CONFIG	
Uncore C-Box Configuration Information (R/O)		Package
3:0	Specifies the number of C-Box units with programmable counters (including processor cores and processor graphics).	
63:4	Reserved.	
Register Address: 2000H, 8192	MSR_UNC_CBO_0_PERFEVTSEL0	
Uncore C-Box 0, Counter 0 Event Select MSR		Package
Register Address: 2001H, 8193	MSR_UNC_CBO_0_PERFEVTSEL1	
Uncore C-Box 0, Counter 1 Event Select MSR		Package
Register Address: 2002H, 8194	MSR_UNC_CBO_0_PERFCTR0	
Uncore C-Box 0, Performance Counter 0		Package
Register Address: 2003H, 8195	MSR_UNC_CBO_0_PERFCTR1	
Uncore C-Box 0, Performance Counter 1		Package
Register Address: 2008H, 8200	MSR_UNC_CBO_1_PERFEVTSEL0	
Uncore C-Box 1, Counter 0 Event Select MSR		Package
Register Address: 2009H, 8201	MSR_UNC_CBO_1_PERFEVTSEL1	
Uncore C-Box 1, Counter 1 Event Select MSR		Package
Register Address: 200AH, 8202	MSR_UNC_CBO_1_PERFCTR0	
Uncore C-Box 1, Performance Counter 0		Package
Register Address: 200BH, 8203	MSR_UNC_CBO_1_PERFCTR1	
Uncore C-Box 1, Performance Counter 1		Package
Register Address: 2010H, 8208	MSR_UNC_CBO_2_PERFEVTSEL0	
Uncore C-Box 2, Counter 0 Event Select MSR		Package
Register Address: 2011H, 8209	MSR_UNC_CBO_2_PERFEVTSEL1	
Uncore C-Box 2, Counter 1 Event Select MSR		Package
Register Address: 2012H, 8210	MSR_UNC_CBO_2_PERFCTR0	
Uncore C-Box 2, Performance Counter 0		Package
Register Address: 2013H, 8211	MSR_UNC_CBO_2_PERFCTR1	
Uncore C-Box 2, Performance Counter 1		Package
Register Address: 2018H, 8216	MSR_UNC_CBO_3_PERFEVTSEL0	
Uncore C-Box 3, Counter 0 Event Select MSR		Package
Register Address: 2019H, 8217	MSR_UNC_CBO_3_PERFEVTSEL1	
Uncore C-Box 3, Counter 1 Event Select MSR		Package
Register Address: 201AH, 8218	MSR_UNC_CBO_3_PERFCTR0	
Uncore C-Box 3, Performance Counter 0		Package
Register Address: 201BH, 8219	MSR_UNC_CBO_3_PERFCTR1	
Uncore C-Box 3, Performance Counter 1		Package

Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 2020H, 8224	MSR_UNC_CBO_4_PERFEVTSEL0	
Uncore C-Box 4, Counter 0 Event Select MSR		Package
Register Address: 2021H, 8225	MSR_UNC_CBO_4_PERFEVTSEL1	
Uncore C-Box 4, Counter 1 Event Select MSR		Package
Register Address: 2022H, 8226	MSR_UNC_CBO_4_PERFCTR0	
Uncore C-Box 4, Performance Counter 0		Package
Register Address: 2023H, 8227	MSR_UNC_CBO_4_PERFCTR1	
Uncore C-Box 4, Performance Counter 1		Package
Register Address: 2028H, 8232	MSR_UNC_CBO_5_PERFEVTSEL0	
Uncore C-Box 5, Counter 0 Event Select MSR		Package
Register Address: 2029H, 8233	MSR_UNC_CBO_5_PERFEVTSEL1	
Uncore C-Box 5, Counter 1 Event Select MSR		Package
Register Address: 202AH, 8234	MSR_UNC_CBO_5_PERFCTR0	
Uncore C-Box 5, Performance Counter 0		Package
Register Address: 202BH, 8235	MSR_UNC_CBO_5_PERFCTR1	
Uncore C-Box 5, Performance Counter 1		Package
Register Address: 2030H, 8240	MSR_UNC_CBO_6_PERFEVTSEL0	
Uncore C-Box 6, Counter 0 Event Select MSR		Package
Register Address: 2031H, 8241	MSR_UNC_CBO_6_PERFEVTSEL1	
Uncore C-Box 6, Counter 1 Event Select MSR		Package
Register Address: 2032H, 8242	MSR_UNC_CBO_6_PERFCTR0	
Uncore C-Box 6, Performance Counter 0		Package
Register Address: 2033H, 8243	MSR_UNC_CBO_6_PERFCTR1	
Uncore C-Box 6, Performance Counter 1		Package
Register Address: 2038H, 8248	MSR_UNC_CBO_7_PERFEVTSEL0	
Uncore C-Box 7, Counter 0 Event Select MSR		Package
Register Address: 2039H, 8249	MSR_UNC_CBO_7_PERFEVTSEL1	
Uncore C-Box 7, Counter 1 Event Select MSR		Package
Register Address: 203AH, 8250	MSR_UNC_CBO_7_PERFCTR0	
Uncore C-Box 7, Performance Counter 0		Package
Register Address: 203BH, 8251	MSR_UNC_CBO_7_PERFCTR1	
Uncore C-Box 7, Performance Counter 1		Package
Register Address: 2040H, 8256	MSR_UNC_CBO_8_PERFEVTSEL0	
Uncore C-Box 8, Counter 0 Event Select MSR		Package
Register Address: 2041H, 8257	MSR_UNC_CBO_8_PERFEVTSEL1	
Uncore C-Box 8, Counter 1 Event Select MSR		Package
Register Address: 2042H, 8258	MSR_UNC_CBO_8_PERFCTR0	

Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Uncore C-Box 8, Performance Counter 0		Package
Register Address: 2043H, 8259	MSR_UNC_CBO_8_PERFCTR1	
Uncore C-Box 8, Performance Counter 1		Package
Register Address: 2048H, 8264	MSR_UNC_CBO_9_PERFEVTSEL0	
Uncore C-Box 9, Counter 0 Event Select MSR		Package
Register Address: 2049H, 8265	MSR_UNC_CBO_9_PERFEVTSEL1	
Uncore C-Box 9, Counter 1 Event Select MSR		Package
Register Address: 204AH, 8266	MSR_UNC_CBO_9_PERFCTR0	
Uncore C-Box 9, Performance Counter 0		Package
Register Address: 204BH, 8267	MSR_UNC_CBO_9_PERFCTR1	
Uncore C-Box 9, Performance Counter 1		Package
Register Address: 2FD0H, 12240	MSR_UNC_ARB_0_PERFEVTSEL0	
Uncore Arb Unit 0, Counter 0 Event Select MSR		Package
Register Address: 2FD1H, 12241	MSR_UNC_ARB_0_PERFEVTSEL1	
Uncore Arb Unit 0, Counter 1 Event Select MSR		Package
Register Address: 2FD2H, 12242	MSR_UNC_ARB_0_PERFCTR0	
Uncore Arb Unit 0, Performance Counter 0		Package
Register Address: 2FD3H, 12243	MSR_UNC_ARB_0_PERFCTR1	
Uncore Arb Unit 0, Performance Counter 1		Package
Register Address: 2FD4H, 12244	MSR_UNC_ARB_0_PERF_STATUS	
Uncore Arb Unit 0, Performance Status		Package
Register Address: 2FD5H, 12245	MSR_UNC_ARB_0_PERF_CTRL	
Uncore Arb Unit 0, Performance Control		Package
Register Address: 2FD8H, 12248	MSR_UNC_ARB_1_PERFEVTSEL0	
Uncore Arb Unit 1, Counter 0 Event Select MSR		Package
Register Address: 2FD9H, 12249	MSR_UNC_ARB_1_PERFEVTSEL1	
Uncore Arb Unit 1, Counter 1 Event Select MSR		Package
Register Address: 2FDAH, 12250	MSR_UNC_ARB_1_PERFCTR0	
Uncore Arb Unit 1, Performance Counter 0		Package
Register Address: 2FDBH, 12251	MSR_UNC_ARB_1_PERFCTR1	
Uncore Arb Unit 1, Performance Counter 1		Package
Register Address: 2FDCH, 12252	MSR_UNC_ARB_1_PERF_STATUS	
Uncore Arb Unit 1, Performance Status		Package
Register Address: 2FDDH, 12253	MSR_UNC_ARB_1_PERF_CTRL	
Uncore Arb Unit 1, Performance Control		Package
Register Address: 2FDEH, 12254	MSR_UNC_PERF_FIXED_CTRL	
Uncore Fixed Counter Control (R/W)		Package

Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
19:0	Reserved.	
20	Enable overflow propagation.	
21	Reserved.	
22	Enable counting.	
63:23	Reserved.	
Register Address: 2FDFH, 12255	MSR_UNC_PERF_FIXED_CTR	
Uncore Fixed Counter		Package
43:0	Current count.	
63:44	Reserved.	
Register Address: 2FF0H, 12272	MSR_UNC_PERF_GLOBAL_CTRL	
Uncore PMU Global Control		Package
0	Slice 0 select.	
1	Slice 1 select.	
2	Slice 2 select.	
3	Slice 3 select.	
4	Slice 4 select.	
18:5	Reserved.	
29	Enable all uncore counters.	
30	Enable wake on PMI.	
31	Enable Freezing counter when overflow.	
63:32	Reserved.	
Register Address: 2FF2H, 12274	MSR_UNC_PERF_GLOBAL_STATUS	
Uncore PMU Main Status		Package
0	Fixed counter overflowed.	
1	An ARB counter overflowed.	
2	Reserved.	
3	A CBox counter overflowed (on any slice).	
63:4	Reserved.	

2.17.6 MSRs Introduced in the Intel® Xeon® Scalable Processor Family

The Intel® Xeon® Scalable Processor Family (CPUID Signature DisplayFamily_DisplayModel value of 06_55H) supports the MSRs listed in Table 2-50.

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Control Features in Intel 64 Processor (R/W) See Table 2-2.		Thread
0	Lock (R/WL)	
1	Enable VMX Inside SMX Operation (R/WL)	
2	Enable VMX Outside SMX Operation (R/WL)	
14:8	SENTER Local Functions Enables (R/WL)	
15	SENTER Global Functions Enable (R/WL)	
18	SGX Global Functions Enable (R/WL)	
20	LMCE_ENABLED (R/WL)	
63:21	Reserved.	
Register Address: 4EH, 78	IA32_PPIN_CTL (MSR_PPIN_CTL)	
Protected Processor Inventory Number Enable Control (R/W)		Package
0	LockOut (R/WO) See Table 2-2.	
1	Enable_PPIN (R/W) See Table 2-2.	
63:2	Reserved.	
Register Address: 4FH, 79	IA32_PPIN (MSR_PPIN)	
Protected Processor Inventory Number (R/O)		Package
63:0	Protected Processor Inventory Number (R/O) See Table 2-2.	
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) See Table 2-26.	Package
22:16	Reserved.	
23	PPIN_CAP (R/O) See Table 2-26.	Package
27:24	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) See Table 2-26.	Package
29	Programmable TDP Limit for Turbo Mode (R/O) See Table 2-26.	Package
30	Programmable TJ OFFSET (R/O) See Table 2-26.	Package
39:31	Reserved.	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
47:40	Maximum Efficiency Ratio (R/O) See Table 2-26.	Package
63:48	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.		Core
2:0	Package C-State Limit (R/W) Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W)	
14:11	Reserved.	
15	CFG Lock (R/WO)	
16	Automatic C-State Conversion Enable (R/W) If 1, the processor will convert HALT or MWAT(C1) to MWAIT(C6).	
24:17	Reserved.	
25	C3 State Auto Demotion Enable (R/W)	
26	C1 State Auto Demotion Enable (R/W)	
27	Enable C3 Undemotion (R/W)	
28	Enable C1 Undemotion (R/W)	
29	Package C State Demotion Enable (R/W)	
30	Package C State Undemotion Enable (R/W)	
63:31	Reserved.	
Register Address: 179H, 377	IA32_MCG_CAP	
Global Machine Check Capability (R/O)		Thread
7:0	Count.	
8	MCG_CTL_P	
9	MCG_EXT_P	
10	MCP_CMCI_P	
11	MCG_TES_P	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
15:12	Reserved.	
23:16	MCG_EXT_CNT	
24	MCG_SER_P	
25	MCG_EM_P	
26	MCG_ELOG_P	
63:27	Reserved.	
Register Address: 17DH, 381	MSR_SMM_MCA_CAP	
Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.		Thread
57:0	Reserved.	
58	SMM_Code_Access_Chk (SMM-RO) If set to 1 indicates that the SMM code access restriction is supported and a host-space interface is available to SMM handler.	
59	Long_Flow_Indication (SMM-RO) If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface is available to SMM handler.	
63:60	Reserved.	
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Core
0	Thermal Status (R/O) See Table 2-2.	
1	Thermal Status Log (R/WC0) See Table 2-2.	
2	PROTCHOT # or FORCEPR# Status (R/O) See Table 2-2.	
3	PROTCHOT # or FORCEPR# Log (R/WC0) See Table 2-2.	
4	Critical Temperature Status (R/O) See Table 2-2.	
5	Critical Temperature Status Log (R/WC0) See Table 2-2.	
6	Thermal Threshold #1 Status (R/O) See Table 2-2.	
7	Thermal Threshold #1 Log (R/WC0) See Table 2-2.	
8	Thermal Threshold #2 Status (R/O) See Table 2-2.	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
9	Thermal Threshold #2 Log (R/WC0) See Table 2-2.	
10	Power Limitation Status (R/O) See Table 2-2.	
11	Power Limitation Log (R/WC0) See Table 2-2.	
12	Current Limit Status (R/O) See Table 2-2.	
13	Current Limit Log (R/WC0) See Table 2-2.	
14	Cross Domain Limit Status (R/O) See Table 2-2.	
15	Cross Domain Limit Log (R/WC0) See Table 2-2.	
22:16	Digital Readout (R/O) See Table 2-2.	
26:23	Reserved.	
30:27	Resolution in Degrees Celsius (R/O) See Table 2-2.	
31	Reading Valid (R/O) See Table 2-2.	
63:32	Reserved.	
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target		Package
15:0	Reserved.	
23:16	Temperature Target (R/O) See Table 2-26.	
27:24	TCC Activation Offset (R/W) See Table 2-26.	
63:28	Reserved.	
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
This register defines the ratio limits. RATIO[0:7] must be populated in ascending order. RATIO[i+1] must be less than or equal to RATIO[i]. Entries with RATIO[i] will be ignored. If any of the rules above are broken, the configuration is silently rejected. If the programmed ratio is: ▪ Above the fused ratio for that core count, it will be clipped to the fuse limits (assuming !OC). ▪ Below the min supported ratio, it will be clipped.		Package
7:0	RATIO_0 Defines ratio limits.	
15:8	RATIO_1 Defines ratio limits.	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
23:16	RATIO_2 Defines ratio limits.	
31:24	RATIO_3 Defines ratio limits.	
39:32	RATIO_4 Defines ratio limits.	
47:40	RATIO_5 Defines ratio limits.	
55:48	RATIO_6 Defines ratio limits.	
63:56	RATIO_7 Defines ratio limits.	
Register Address: 1AEH, 430	MSR_TURBO_RATIO_LIMIT_CORES	
This register defines the active core ranges for each frequency point. NUMCORE[0:7] must be populated in ascending order. NUMCORE[i+1] must be greater than NUMCORE[i]. Entries with NUMCORE[i] == 0 will be ignored. The last valid entry must have NUMCORE >= the number of cores in the SKU. If any of the rules above are broken, the configuration is silently rejected.		Package
7:0	NUMCORE_0 Defines the active core ranges for each frequency point.	
15:8	NUMCORE_1 Defines the active core ranges for each frequency point.	
23:16	NUMCORE_2 Defines the active core ranges for each frequency point.	
31:24	NUMCORE_3 Defines the active core ranges for each frequency point.	
39:32	NUMCORE_4 Defines the active core ranges for each frequency point.	
47:40	NUMCORE_5 Defines the active core ranges for each frequency point.	
55:48	NUMCORE_6 Defines the active core ranges for each frequency point.	
63:56	NUMCORE_7 Defines the active core ranges for each frequency point.	
Register Address: 280H, 640	IA32_MC0_CTL2	
See Table 2-2.		Core
Register Address: 281H, 641	IA32_MC1_CTL2	
See Table 2-2.		Core
Register Address: 282H, 642	IA32_MC2_CTL2	
See Table 2-2.		Core

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 283H, 643	IA32_MC3_CTL2	
See Table 2-2.		Core
Register Address: 284H, 644	IA32_MC4_CTL2	
See Table 2-2.		Package
Register Address: 285H, 645	IA32_MC5_CTL2	
See Table 2-2.		Package
Register Address: 286H, 646	IA32_MC6_CTL2	
See Table 2-2.		Package
Register Address: 287H, 647	IA32_MC7_CTL2	
See Table 2-2.		Package
Register Address: 288H, 648	IA32_MC8_CTL2	
See Table 2-2.		Package
Register Address: 289H, 649	IA32_MC9_CTL2	
See Table 2-2.		Package
Register Address: 28AH, 650	IA32_MC10_CTL2	
See Table 2-2.		Package
Register Address: 28BH, 651	IA32_MC11_CTL2	
See Table 2-2.		Package
Register Address: 28CH, 652	IA32_MC12_CTL2	
See Table 2-2.		Package
Register Address: 28DH, 653	IA32_MC13_CTL2	
See Table 2-2.		Package
Register Address: 28EH, 654	IA32_MC14_CTL2	
See Table 2-2.		Package
Register Address: 28FH, 655	IA32_MC15_CTL2	
See Table 2-2.		Package
Register Address: 290H, 656	IA32_MC16_CTL2	
See Table 2-2.		Package
Register Address: 291H, 657	IA32_MC17_CTL2	
See Table 2-2.		Package
Register Address: 292H, 658	IA32_MC18_CTL2	
See Table 2-2.		Package
Register Address: 293H, 659	IA32_MC19_CTL2	
See Table 2-2.		Package
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC0 reports MC errors from the IFU module.		Core

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC0 reports MC errors from the IFU module.		Core
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC0 reports MC errors from the IFU module.		Core
Register Address: 403H, 1027	IA32_MC0_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC0 reports MC errors from the IFU module.		Core
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.		Core
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.		Core
Register Address: 406H, 1030	IA32_MC1_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.		Core
Register Address: 407H, 1031	IA32_MC1_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.		Core
Register Address: 408H, 1032	IA32_MC2_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.		Core
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.		Core
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.		Core
Register Address: 40BH, 1035	IA32_MC2_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.		Core
Register Address: 40CH, 1036	IA32_MC3_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC3 reports MC errors from the MLC module.		Core
Register Address: 40DH, 1037	IA32_MC3_STATUS	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC3 reports MC errors from the MLC module.		Core
Register Address: 40EH, 1038	IA32_MC3_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC3 reports MC errors from the MLC module.		Core
Register Address: 40FH, 1039	IA32_MC3_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC3 reports MC errors from the MLC module.		Core
Register Address: 410H, 1040	IA32_MC4_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC4 reports MC errors from the PCU module.		Package
Register Address: 411H, 1041	IA32_MC4_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC4 reports MC errors from the PCU module.		Package
Register Address: 412H, 1042	IA32_MC4_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC4 reports MC errors from the PCU module.		Package
Register Address: 413H, 1043	IA32_MC4_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC4 reports MC errors from the PCU module.		Package
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from a link interconnect module.		Package
Register Address: 415H, 1045	IA32_MC5_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from a link interconnect module.		Package
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from a link interconnect module.		Package
Register Address: 417H, 1047	IA32_MC5_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC5 reports MC errors from a link interconnect module.		Package
Register Address: 418H, 1048	IA32_MC6_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 419H, 1049	IA32_MC6_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 41AH, 1050	IA32_MC6_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41BH, 1051	IA32_MC6_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC6 reports MC errors from the integrated I/O module.		Package
Register Address: 41CH, 1052	IA32_MC7_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the M2M 0.		Package
Register Address: 41DH, 1053	IA32_MC7_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the M2M 0.		Package
Register Address: 41EH, 1054	IA32_MC7_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the M2M 0.		Package
Register Address: 41FH, 1055	IA32_MC7_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC7 reports MC errors from the M2M 0.		Package
Register Address: 420H, 1056	IA32_MC8_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the M2M 1.		Package
Register Address: 421H, 1057	IA32_MC8_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the M2M 1.		Package
Register Address: 422H, 1058	IA32_MC8_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the M2M 1.		Package
Register Address: 423H, 1059	IA32_MC8_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC8 reports MC errors from the M2M 1.		Package
Register Address: 424H, 1060	IA32_MC9_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 425H, 1061	IA32_MC9_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 426H, 1062	IA32_MC9_ADDR	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 427H, 1063	IA32_MC9_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 428H, 1064	IA32_MC10_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 429H, 1065	IA32_MC10_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 42AH, 1066	IA32_MC10_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 42BH, 1067	IA32_MC10_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 42CH, 1068	IA32_MC11_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 42DH, 1069	IA32_MC11_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 42EH, 1070	IA32_MC11_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 42FH, 1071	IA32_MC11_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC9 - MC11 report MC errors from the CHA.		Package
Register Address: 430H, 1072	IA32_MC12_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC12 report MC errors from each channel of a link interconnect module.		Package
Register Address: 431H, 1073	IA32_MC12_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC12 report MC errors from each channel of a link interconnect module.		Package
Register Address: 432H, 1074	IA32_MC12_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC12 report MC errors from each channel of a link interconnect module.		Package

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 433H, 1075	IA32_MC12_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC12 report MC errors from each channel of a link interconnect module.		Package
Register Address: 434H, 1076	IA32_MC13_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 435H, 1077	IA32_MC13_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 436H, 1078	IA32_MC13_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 437H, 1079	IA32_MC13_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 438H, 1080	IA32_MC14_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 439H, 1081	IA32_MC14_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 43AH, 1082	IA32_MC14_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 43BH, 1083	IA32_MC14_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 43CH, 1084	IA32_MC15_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 43DH, 1085	IA32_MC15_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 43EH, 1086	IA32_MC15_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 43FH, 1087	IA32_MC15_MISC	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 440H, 1088	IA32_MC16_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 441H, 1089	IA32_MC16_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 442H, 1090	IA32_MC16_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 443H, 1091	IA32_MC16_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 444H, 1092	IA32_MC17_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 445H, 1093	IA32_MC17_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 446H, 1094	IA32_MC17_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 447H, 1095	IA32_MC17_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 448H, 1096	IA32_MC18_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 449H, 1097	IA32_MC18_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 44AH, 1098	IA32_MC18_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package
Register Address: 44BH, 1099	IA32_MC18_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Banks MC13 through MC 18 report MC errors from the integrated memory controllers.		Package

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 44CH, 1100	IA32_MC19_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a link interconnect module.		Package
Register Address: 44DH, 1101	IA32_MC19_STATUS	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a link interconnect module.		Package
Register Address: 44EH, 1102	IA32_MC19_ADDR	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a link interconnect module.		Package
Register Address: 44FH, 1103	IA32_MC19_MISC	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs," through Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." Bank MC19 reports MC errors from a link interconnect module.		Package
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers Used in RAPL Interfaces (R/O)		Package
3:0	Power Units See Section 16.10.1, "RAPL Interfaces."	Package
7:4	Reserved.	Package
12:8	Energy Status Units Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).	Package
15:13	Reserved.	Package
19:16	Time Units See Section 16.10.1, "RAPL Interfaces."	Package
63:20	Reserved.	
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
DRAM RAPL Power Limit Control (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) Energy consumed by DRAM devices.		Package
31:0	Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).	
63:32	Reserved.	
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
DRAM RAPL Parameters (R/W) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 620H, 1568	MSR_UNCORE_RATIO_LIMIT	
Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.		Package
63:15	Reserved.	
14:8	MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.	
7	Reserved.	
6:0	MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.	
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
Reserved (R/O) Reads return 0.		Package
Register Address: C8DH, 3213	IA32_QM_EVTSEL	
Monitoring Event Select Register (R/W) If CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1.		Thread
7:0	EventID (R/W) Event encoding: 0x00: No monitoring. 0x01: L3 occupancy monitoring. 0x02: Total memory bandwidth monitoring. 0x03: Local memory bandwidth monitoring. All other encoding reserved.	
31:8	Reserved.	
41:32	RMID (R/W)	
63:42	Reserved.	
Register Address: C8FH, 3215	IA32_PQR_ASSOC	
Resource Association Register (R/W)		Thread
9:0	RMID	
31:10	Reserved.	
51:32	CLOS (R/W)	
63: 52	Reserved.	
Register Address: C90H, 3216	IA32_L3_QOS_MASK_0	
L3 Class Of Service Mask - CLOS 0 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 0 enforcement.	
63:20	Reserved.	

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: C91H, 3217	IA32_L3_QOS_MASK_1	
L3 Class Of Service Mask - CLOS 1 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 1 enforcement.	
63:20	Reserved.	
Register Address: C92H, 3218	IA32_L3_QOS_MASK_2	
L3 Class Of Service Mask - CLOS 2 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 2 enforcement.	
63:20	Reserved.	
Register Address: C93H, 3219	IA32_L3_QOS_MASK_3	
L3 Class Of Service Mask - CLOS 3 (R/W). If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 3 enforcement.	
63:20	Reserved.	
Register Address: C94H, 3220	IA32_L3_QOS_MASK_4	
L3 Class Of Service Mask - CLOS 4 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 4 enforcement.	
63:20	Reserved.	
Register Address: C95H, 3221	IA32_L3_QOS_MASK_5	
L3 Class Of Service Mask - CLOS 5 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 5 enforcement.	
63:20	Reserved.	
Register Address: C96H, 3222	IA32_L3_QOS_MASK_6	
L3 Class Of Service Mask - CLOS 6 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 6 enforcement.	
63:20	Reserved.	
Register Address: C97H, 3223	IA32_L3_QOS_MASK_7	
L3 Class Of Service Mask - CLOS 7 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 7 enforcement.	
63:20	Reserved.	
Register Address: C98H, 3224	IA32_L3_QOS_MASK_8	
L3 Class Of Service Mask - CLOS 8 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8.		Package

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

Register Address: Hex, Decimal	Register Name (Former Register Name)	
Register Information / Bit Fields	**Bit Description**	**Scope**
0:19	CBM: Bit vector of available L3 ways for CLOS 8 enforcement.	
63:20	Reserved.	
Register Address: C99H, 3225	IA32_L3_QOS_MASK_9	
L3 Class Of Service Mask - CLOS 9 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 9 enforcement.	
63:20	Reserved.	
Register Address: C9AH, 3226	IA32_L3_QOS_MASK_10	
L3 Class Of Service Mask - CLOS 10 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 10 enforcement.	
63:20	Reserved.	
Register Address: C9BH, 3227	IA32_L3_QOS_MASK_11	
L3 Class Of Service Mask - CLOS 11 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=11.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 11 enforcement.	
63:20	Reserved.	
Register Address: C9CH, 3228	IA32_L3_QOS_MASK_12	
L3 Class Of Service Mask - CLOS 12 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=12.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 12 enforcement.	
63:20	Reserved.	
Register Address: C9DH, 3229	IA32_L3_QOS_MASK_13	
L3 Class Of Service Mask - CLOS 13 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=13.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 13 enforcement.	
63:20	Reserved.	
Register Address: C9EH, 3230	IA32_L3_QOS_MASK_14	
L3 Class Of Service Mask - CLOS 14 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 14 enforcement.	
63:20	Reserved.	
Register Address: C9FH, 3231	IA32_L3_QOS_MASK_15	
L3 Class Of Service Mask - CLOS 15 (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15.		Package
0:19	CBM: Bit vector of available L3 ways for CLOS 15 enforcement.	
63:20	Reserved.	

2.17.7 MSRs Specific to the 3rd Generation Intel® Xeon® Scalable Processor Family Based on Ice Lake Microarchitecture

The 3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_6AH or 06_6CH) support the MSRs listed in Table 2-51.

Table 2-51. MSRs Supported by the 3rd Generation Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_6AH or 06_6CH

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 612H, 1554	MSR_PACKAGE_ENERGY_TIME_STATUS	
Package energy consumed by the entire CPU (R/W)		Package
31:0	Total amount of energy consumed since last reset.	
63:32	Total time elapsed when the energy was last updated. This is a monotonic increment counter with auto wrap back to zero after overflow. Unit is 10ns.	
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
Allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.		Package
14:0	DRAM_PP_PWR_LIM: Power Limit[0] for DDR domain. Units = Watts, Format = 11.3, Resolution = 0.125W, Range = 0-2047.875W.	
15	PWR_LIM_CTRL_EN: Power Limit[0] enable bit for DDR domain.	
16	Reserved.	
23:17	CTRL_TIME_WIN: Power Limit[0] time window Y value, for DDR domain. Actual time_window for RAPL is: (1/1024 seconds) * (1+(x/4)) * (2^y)	
62:24	Reserved.	
63	PP_PWR_LIM_LOCK: When set, this entire register becomes read-only. This bit will typically be set by BIOS during boot.	
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
31:0	Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).	
63:32	Reserved.	
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, "DRAM RAPL Domain."		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	
DRAM Power Parameters (R/W)		Package

Table 2-51. MSRs Supported by the 3rd Generation Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_6AH or 06_6CH (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
14:0	Spec DRAM Power (DRAM_TDP): The Spec power allowed for DRAM. The TDP setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].	
15	Reserved.	
30:16	Minimal DRAM Power (DRAM_MIN_PWR): The minimal power setting allowed for DRAM. Lower values will be clamped to this value. The minimum setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].	
31	Reserved.	
46:32	Maximal Package Power (DRAM_MAX_PWR): The maximal power setting allowed for DRAM. Higher values will be clamped to this value. The maximum setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].	
47	Reserved.	
54:48	Maximal Time Window (DRAM_MAX_WIN): The maximal time window allowed for the DRAM. Higher values will be clamped to this value. x = PKG_MAX_WIN[54:53] y = PKG_MAX_WIN[52:48] The timing interval window is a floating-point number given by 1.x *power(2,y). The unit of measurement is defined in MSR_DRAM_POWER_INFO_UNIT[TIME_UNIT].	
62:55	Reserved.	
63	LOCK: Lock bit to lock the register.	
Register Address: 981H, 2433	IA32_TME_CAPABILITY	
See Table 2-2.		
Register Address: 982H, 2434	IA32_TME_ACTIVATE	
See Table 2-2.		
Register Address: 983H, 2435	IA32_TME_EXCLUDE_MASK	
See Table 2-2.		
Register Address: 984H, 2436	IA32_TME_EXCLUDE_BASE	
See Table 2-2.		

2.17.8 MSRs Specific to the 4th and 5th Generation Intel® Xeon® Scalable Processor Families

The 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_8FH) and the 5th generation Intel® Xeon® Scalable Processor Family based on Emerald Rapids microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_CFH) both support the MSRs listed in Section 2.17, "MSRs In the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, Intel® Xeon® E Processors, Intel® Xeon® 6 P-core processors, Intel® Xeon® 6 E-core processors, and Intel® Series 2 Core™ Ultra Processors," including Table 2-52. For an MSR listed in Table 2-52 that also appears in the model-specific tables of prior generations, Table 2-52 supersedes prior generation tables.

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 33H, 51	MSR_MEMORY_CTRL	
Memory Control Register (R/W)		Core
27:0	Reserved.	
28	UC_LOCK_DISABLE If set to 1, a UC lock will cause a #GP(0) exception. See Section 10.1.2.3, "Features to Disable Bus Locks."	
29	SPLIT_LOCK_DISABLE If set to 1, a split lock will cause an #AC(0) exception. See Section 10.1.2.3, "Features to Disable Bus Locks."	
31:30	Reserved.	
Register Address: A7H, 167	MSR_BIOS_DEBUG	
BIOS DEBUG (R/O) See Table 2-45.		Thread
Register Address: BCH, 188	IA32_MISC_PACKAGE_CTLS	
Power Filtering Control (R/W) IA32_ARCH_CAPABILITIES[bit 10] enumerates support for this MSR. See Table 2-2.		Package
Register Address: CFH, 207	IA32_CORE_CAPABILITIES	
IA32 Core Capabilities Register (R/W) If CPUID.(EAX=07H, ECX=0):EDX[30] = 1. This MSR provides an architectural enumeration function for model-specific behavior.		Core
0	Reserved: returns zero.	
1	Reserved: returns zero.	
2	INTEGRITY_CAPABILITIES When set to 1, the processor supports MSR_INTEGRITY_CAPABILITIES.	
3	RSM_IN_CPL0_ONLY Indicates that RSM will only be allowed in CPL0 and will #GP for all non-CPL0 privilege levels.	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
4	UC_LOCK_DISABLE_SUPPORTED When read as 1, software can set bit 28 of MSR_MEMORY_CTRL (MSR address 33H).	
5	SPLIT_LOCK_DISABLE_SUPPORTED When read as 1, software can set bit 29 of MSR_MEMORY_CTRL.	
6	Reserved: returns zero.	
7	UC_STORE_THROTTLING_SUPPORTED Indicates that the snoop filter quality of service MSRs are supported on this core. This is based on the existence of a non-inclusive cache and the L2/MLC QoS feature supported.	
63:8	Reserved: returns zero.	
Register Address: E1H, 225	IA32_UMWAIT_CONTROL	
UMWAIT Control (R/W) See Table 2-2.		
Register Address: EDH, 237	MSR_RAR_CONTROL	
RAR Control (R/W)		Thread
63:32	Reserved.	
31	ENABLE RAR events are recognized. When RAR is not enabled, RARs are dropped.	
30	IGNORE_IF Allow RAR servicing at the RLP regardless of the value of RFLAGS.IF.	
29:0	Reserved.	
Register Address: EEH, 238	MSR_RAR_ACTION_VECTOR_BASE	
Pointer to RAR Action Vector (R/W)		Thread
63:MAXPHYADDR	Reserved.	
MAXPHYADDR-1:6	VECTOR_PHYSICAL_ADDRESS Pointer to the physical address of the 64B aligned RAR action vector.	
5:0	Reserved.	
Register Address: EFH, 239	MSR_RAR_PAYLOAD_TABLE_BASE	
Pointer to Base of RAR Payload Table (R/W)		Thread
63:MAXPHYADDR	Reserved.	
MAXPHYADDR-1:12	TABLE_PHYSICAL_ADDRESS Pointer to the base physical address of the 4K aligned RAR payload table.	
11:0	Reserved.	
Register Address: F0H, 240	MSR_RAR_INFO	
Read Only RAR Information (RO)		Thread
63:38	Always zero.	
37:32	Table Max Index Maximum supported payload table index.	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:0	Supported payload type bitmap. A value of 1 in bit position [i] indicates that payload type [i] is supported.	
Register Address: 105H, 261	MSR_CORE_BIST	
Core BIST (R/W) Controls Array BIST activation and status checking as part of FUSA.		Core
31:0	BIST_ARRAY Bitmap indicating which arrays to run BIST on (WRITE). Bitmap indicating which arrays were not processed, i.e., completion mask (READ).	
39:32	BANK Array bank of the [least significant set bit] array indicated in EAX to start BIST(WRITE). Array bank interrupted or failed (READ).	
47:40	DWORD Array dword of the [least significant set bit] array indicated in EAX to start BIST (WRITE). Array dword interrupted or failed (READ).	
62:48	Reserved.	
63	CTRL_RESULT Indicates whether WRMSR should signal Machine-Check upon BIST-error (WRITE). BIST result PASS(0)/FAIL(1) of the (least significant set bit) array indicated in EAX (READ).	
Register Address: 10AH, 266	IA32_ARCH_CAPABILITIES	
Enumeration of Architectural Features (R/O) See Table 2-2.		
Register Address: 1A4H, 420	MSR_PREFETCH_CONTROL	
Prefetch Disable Bits (R/W)		
0	L2_HARDWARE_PREFETCHER_DISABLE If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.	
1	L2_ADJACENT_CACHE_LINE_PREFETCHER_DISABLE If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).	
2	DCU_HARDWARE_PREFETCHER_DISABLE If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.	
3	DCU_IP_PREFETCHER_DISABLE If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.	
4	Reserved.	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
5	AMP_PREFETCH_DISABLE If 1, disables the L2 Adaptive Multipath Probability (AMP) prefetcher.	
63:6	Reserved.	
Register Address: 1ADH, 429	MSR_PRIMARY_TURBO_RATIO_LIMIT	
Primary Maximum Turbo Ratio Limit (R/W) See Table 2-46.		Package
Register Address: 1AEH, 430	MSR_TURBO_RATIO_LIMIT_CORES	
See Table 2-50.		Package
Register Address: 1C4H, 452	IA32_XFD	
Extended Feature Detect (R/W) See Table 2-2.		
Register Address: 1C5H, 453	IA32_XFD_ERR	
XFD Error Code (R/W) See Table 2-2.		
Register Address: 2C2H, 706	MSR_COPY_SCAN_HASHES	
COPY_SCAN_HASHES (W)		Die
63:0	SCAN_HASH_ADDR Contains the linear address of the SCAN Test HASH Binary loaded into memory.	
Register Address: 2C3H, 707	MSR_SCAN_HASHES_STATUS	
SCAN_HASHES_STATUS (R/O)		
15:0	CHUNK_SIZE Chunk size of the test in KB.	Die
23:16	NUM_CHUNKS Total number of chunks.	Die
31:24	Reserved: all zeros.	
39:32	ERROR_CODE The error-code refers to the LP that runs WRMSR(2C2H). 0x0: No error reported. 0x1: Attempt to copy scan-hashes when copy already in progress. 0x2: Secure Memory not set up correctly. 0x3: Scan-image header Image_info.ProgramID doesn't match RDMSR(2D9H)[31:24], or scan-image header Processor-Signature doesn't match F/M/S, or scan-image header Processor-Flags doesn't match PlatformID. 0x4: Reserved 0x5: Integrity check failed. 0x6: Re-install of scan test image attempted when current scan test image is in use by other LPs.	Thread
50:40	Reserved: set to all zeros.	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
62:51	MAX_CORE_LIMIT Maximum Number of cores that can run Intel® In-field Scan simultaneously minus 1. 0 means 1 core at a time.	Die
63	Valid Valid bit is set when COPY_SCAN_HASHES has completed successfully.	Die
Register Address: 2C4H, 708	MSR_AUTHENTICATE_AND_COPY_CHUNK	
AUTHENTICATE_AND_COPY_CHUNK (W)		Die
7:0	CHUNK_INDEX Chunk Index, should be less than the total number of chunks defined by NUM_CHUNKS (MSR_SCAN_HASHES_STATUS[23:16]).	
63:8	CHUNK_ADDR Bits 63:8 of 256B aligned Linear address of scan chunk in memory.	
Register Address: 2C5H, 709	MSR_CHUNKS_AUTHENTICATION_STATUS	
CHUNKS_AUTHENTICATION_STATUS (R/O)		
7:0	VALID_CHUNKS Total number of Valid (authenticated) chunks.	Die
15:8	TOTAL_CHUNKS Total number of chunks.	Die
31:16	Reserved: all zeros.	
39:32	ERROR_CODE The error code refers to the LP that runs WRMSR(2C4H). 0x0: No error reported. 0x1: Attempt to authenticate a CHUNK which is already marked as authentic or is currently being installed by another core. 0x2: CHUNK authentication error. HASH of chunk did not match expected value.	Thread
63:40	Reserved: set to all zeros.	
Register Address: 2C6H, 710	MSR_ACTIVATE_SCAN	
ACTIVATE_SCAN (W)		Thread
7:0	CHUNK_START_INDEX Indicates chunk index to start from.	
15:8	CHUNK_STOP_INDEX Indicates what chunk index to stop at (inclusive).	
31:16	Reserved: all zeros.	
62:32	THREAD_WAIT_DELAY TSC based delay to allow threads to rendezvous.	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
63	SIGNAL_MCE If 1, then on scan-error log MC in MC4_STATUS and signal MCE if machine check signaling enabled in MC4_CTL[0]. If 0, then no logging/no signaling.	
Register Address: 2C7H, 711	MSR_SCAN_STATUS	
SCAN_STATUS (R/O)		
7:0	CHUNK_NUM SCAN Chunk that was reached.	Core
15:8	CHUNK_STOP_INDEX Indicates what chunk index to stop at (inclusive). Maps to same field in WRMSR(ACTIVATE_SCAN).	Core
31:16	Reserved: return all zeros.	
39:32	ERROR_CODE 0x0: No error. 0x1: SCAN operation did not start. Other thread did not join in time. 0x2: SCAN operation did not start. Interrupt occurred prior to threads rendezvous. 0x3: SCAN operation did not start. Power Management conditions are inadequate to run Intel In-field Scan. 0x4: SCAN operation did not start. Non-valid chunks in the range CHUNK_STOP_INDEX : CHUNK_START_INDEX. 0x5: SCAN operation did not start. Mismatch in arguments between threads T0/T1. 0x6: SCAN operation did not start. Core not capable of performing SCAN currently. 0x8: SCAN operation did not start. Exceeded number of Logical Processors (LP) allowed to run Intel In-field Scan concurrently. MAX_CORE_LIMIT exceeded. 0x9: Interrupt occurred. Scan operation aborted prematurely, not all chunks requested have been executed.	Thread
61:40	Reserved: return all zeros.	
62	SCAN_CONTROL_ERROR Scan-System-Controller malfunction.	Core
63	SCAN_SIGNATURE_ERROR Core failed SCAN-SIGNATURE checking for this chunk.	Core
Register Address: 2C8H, 712	MSR_SCAN_MODULE_ID	
SCAN_MODULE_ID (R/O)		Module
31:0	RevID of the currently installed scan test image. Maps to Revision field in external header (offset 4).	
63:32	Reserved: return all zeros.	
Register Address: 2C9H, 713	MSR_LAST_SAF_WP	
LAST_SAF_WP (R/O)		Core

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:0	LAST_WP Provides information about the core when the last WRMSR(ACTIVATE_SCAN) was executed. Available only if enumerated in MSR_INTEGRITY_CAPABILITIES[10:9].	
63:32	Reserved: return all zeros.	
Register Address: 2D9H, 729	MSR_INTEGRITY_CAPABILITIES	
INTEGRITY_CAPABILITIES (R/O)		Module
0	STARTUP_SCAN_BIST When set, supports Intel In-field Scan.	
3:1	Reserved: return all zeros.	
4	PERIODIC_SCAN_BIST When set, supports Intel In-field Scan.	
23:5	Reserved: return all zeros.	
31:24	ID of the scan programs supported for this part. WRMSR(2C2H) verifies this value against the corresponding value in the scan-image header, i.e., Image_info.	
Register Address: 410H, 1040	IA32_MC4_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC4 reports MC errors from the PCU module. If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.		Package
Register Address: 411H, 1041	IA32_MC4_STATUS	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC4 reports MC errors from the PCU module. If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.		Package
Register Address: 412H, 1042	IA32_MC4_ADDR	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC4 reports MC errors from the PCU module. If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.		Package
Register Address: 413H, 1043	IA32_MC4_MISC	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs,” through Section 17.3.2.4, “IA32_MC**i**_MISC MSRs.” Bank MC4 reports MC errors from the PCU module. If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.		Package
Register Address: 492H, 1170	IA32_VMX_PROCBASED_CTLS3	
Capability Reporting Register of Tertiary Processor-Based VM-Execution Controls (R/O) See Table 2-2.		
Register Address: 493H, 1171	IA32_VMX_EXIT_CTLS2	
Capability Reporting Register of Secondary VM-Exit Controls (R/O) See Table 2-2.		
Register Address: 540H, 1344	MSR_THREAD_UARCH_CTL	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Thread Microarchitectural Control (R/W) See Table 2-47.		Thread
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) Energy consumed by DRAM devices.		Package
31:0	Energy in 61 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).	
63:32	Reserved.	
Register Address: 64DH, 1613	MSR_PLATFORM_ENERGY_STATUS	
Platform Energy Status (R/O)		Package
31:0	TOTAL_ENERGY_CONSUMED Total energy consumption in J (32.0), in 10nsec units.	
63:32	TIME_STAMP Time stamp (U32.0).	
Register Address: 65CH, 1628	MSR_PLATFORM_POWER_LIMIT	
Platform Power Limit Control (R/W-L)		Package
16:0	POWER_LIMIT_1 The average power limit value that the platform must not exceed over a time window as specified by the Power_Limit_1_TIME field. The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPL_POWER_UNIT.	
17	POWER_LIMIT_1_EN When set, the processor can apply control policies such that the platform average power does not exceed the Power_Limit_1 value over an exponential weighted moving average of the time window.	
18	CRITICAL_POWER_CLAMP_1 When set, the processor can go below the OS-requested P States to maintain the power below the specified Power_Limit_1 value.	
25:19	POWER_LIMIT_1_TIME This indicates the time window over which the Power_Limit_1 value should be maintained. This field is made up of two numbers from the following equation: Time Window = (float) ((1+(X/4))*(2^Y)), where: X = POWER_LIMIT_1_TIME[23:22] Y = POWER_LIMIT_1_TIME[21:17] The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is 0DH, and the unit is specified in MSR_RAPL_POWER_UNIT[Time Unit].	
31:26	Reserved.	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
48:32	POWER_LIMIT_2 This is the Duration Power limit value that the platform must not exceed. The unit is specified in MSR_RAPL_POWER_UNIT.	
49	Enable Platform Power Limit #2 When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #2 over the Short Duration time window.	
50	Platform Clamping Limitation #2 When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.	
57:51	POWER_LIMIT_2_TIME This indicates the time window over which the Power_Limit_2 value should be maintained. This field has the same format as the POWER_LIMIT_1_TIME field.	
62:58	Reserved.	
63	LOCK Setting this bit will lock all other bits of this MSR until system RESET.	
Register Address: 665H, 1637	MSR_PLATFORM_POWER_INFO	
Platform Power Information (R/W)		Package
16:0	MAX_PPL1 Maximum PP L1 value. The unit is specified in MSR_RAPL_POWER_UNIT.	
31:17	MIN_PPL1 Minimum PP L1 value. The unit is specified in MSR_RAPL_POWER_UNIT.	
48:32	MAX_PPL2 Maximum PP L2 value. The unit is specified in MSR_RAPL_POWER_UNIT.	
55:49	MAX_TW Maximum time window. The unit is specified in MSR_RAPL_POWER_UNIT.	
62:56	Reserved.	
63	LOCK Setting this bit will lock all other bits of this MSR until system RESET.	
Register Address: 666H, 1638	MSR_PLATFORM_RAPL_SOCKET_PERF_STATUS	
Platform RAPL Socket Performance Status (R/O)		Package
31:0	Count of limited performance due to platform RAPL limit.	
Register Address: 6A0H, 1696	IA32_U_CET	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Configure User Mode CET (R/W) See Table 2-2.		
Register Address: 6A2H, 1698	IA32_S_CET	
Configure Supervisor Mode CET (R/W) See Table 2-2.		
Register Address: 6A4H, 1700	IA32_PL0_SSP	
Linear address to be loaded into SSP on transition to privilege level 0. (R/W) See Table 2-2.		
Register Address: 6A5H, 1701	IA32_PL1_SSP	
Linear address to be loaded into SSP on transition to privilege level 1. (R/W) See Table 2-2.		
Register Address: 6A6H, 1702	IA32_PL2_SSP	
Linear address to be loaded into SSP on transition to privilege level 2. (R/W) See Table 2-2.		
Register Address: 6A7H, 1703	IA32_PL3_SSP	
Linear address to be loaded into SSP on transition to privilege level 3. (R/W) See Table 2-2.		
Register Address: 6A8H, 1704	IA32_INTERRUPT_SSP_TABLE_ADDR	
Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W) See Table 2-2.		
Register Address: 6E1H, 1761	IA32_PKRS	
Specifies the PK permissions associated with each protection domain for supervisor pages (R/W) See Table 2-2.		
Register Address: 776H, 1910	IA32_HWP_CTL	
See Table 2-2.		
Register Address: 981H, 2433	IA32_TME_CAPABILITY	
Memory Encryption Capability MSR See Table 2-2.		
Register Address: 985H, 2437	IA32_UINTR_RR	
User Interrupt Request Register (R/W) See Table 2-2.		
Register Address: 986H, 2438	IA32_UINTR_HANDLER	
User Interrupt Handler Address (R/W) See Table 2-2.		
Register Address: 987H, 2439	IA32_UINTR_STACKADJUST	
User Interrupt Stack Adjustment (R/W) See Table 2-2.		
Register Address: 988H, 2440	IA32_UINTR_MISC	

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
User-Interrupt Target-Table Size and Notification Vector (R/W) See Table 2-2.		
Register Address: 989H, 2441	IA32_UINTR_PD	
User Interrupt PID Address (R/W) See Table 2-2.		
Register Address: 98AH, 2442	IA32_UINTR_TT	
User-Interrupt Target Table (R/W) See Table 2-2.		
Register Address: C70H, 3184	MSR_B1_PMON_EVNT_SEL0	
Uncore B-box 1 PerfMon event select MSR.		Package
Register Address: C71H, 3185	MSR_B1_PMON_CTR0	
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C72H, 3186	MSR_B1_PMON_EVNT_SEL1	
Uncore B-box 1 PerfMon event select MSR.		Package
Register Address: C73H, 3187	MSR_B1_PMON_CTR1	
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C74H, 3188	MSR_B1_PMON_EVNT_SEL2	
Uncore B-box 1 PerfMon event select MSR.		Package
Register Address: C75H, 3189	MSR_B1_PMON_CTR2	
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C76H, 3190	MSR_B1_PMON_EVNT_SEL3	
Uncore B-box 1vPerfMon event select MSR.		Package
Register Address: C77H, 3191	MSR_B1_PMON_CTR3	
Uncore B-box 1 PerfMon counter MSR.		Package
Register Address: C82H, 3122	MSR_W_PMON_BOX_OVF_CTRL	
Uncore W-box PerfMon local box overflow control MSR.		Package
Register Address: C8FH, 3215	IA32_PQR_ASSOC	
See Table 2-2.		
Register Address: C90H—C9EH, 3216—3230	IA32_L3_QOS_MASK_0 through IA32_L3_QOS_MASK_14	
See Table 2-50.		Package
Register Address: D10H—D17H, 3344—3351	IA32_L2_QOS_MASK_[0-7]	
IA32_CR_L2_QOS_MASK_[0-7] If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] ≥ 0. See Table 2-2.		Core
Register Address: D93H, 3475	IA32_PASID	
See Table 2-2.		

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1200H—121FH, 4608—4639	IA32_LBR_x_INFO	
Last Branch Record Entry X Info Register (R/W) See Table 2-2.		
Register Address: 1406H, 5126	IA32_MCU_CONTROL	
See Table 2-2.		
Register Address: 14CEH, 5326	IA32_LBR_CTL	
Last Branch Record Enabling and Configuration Register (R/W) See Table 2-2.		
Register Address: 14CFH, 5327	IA32_LBR_DEPTH	
Last Branch Record Maximum Stack Depth Register (R/W) See Table 2-2.		
Register Address: 1500H—151FH, 5376—5407	IA32_LBR_x_FROM_IP	
Last Branch Record Entry X Source IP Register (R/W) See Table 2-2.		
Register Address: 1600H—161FH, 5632—5663	IA32_LBR_x_TO_IP	
Last Branch Record Entry X Destination IP Register (R/W) See Table 2-2.		

2.17.9 MSRs Introduced in the Intel® Core™ Ultra 7 Processor Supporting Performance Hybrid Architecture

Table 2-53 lists additional MSRs for the Intel Core Ultra 7 processor with a CPUID Signature DisplayFamily_Display-Model value of 06_AAH. Table 2-54 lists the MSRs unique to the processor P-core. Table 2-55 lists the MSRs unique to the processor E-core.

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 33H, 51	MSR_MEMORY_CTRL	
Memory Control Register		Core
26:0	Reserved.	
27	UC_STORE_THROTTLE If set to 1, when enabled, the processor will only allow one in-progress UC store at a time.	
28	UC_LOCK_DISABLE If set to 1, a UC lock will cause a #GP(0) exception. See Section 10.1.2.3, “Features to Disable Bus Locks.”	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
29	SPLIT_LOCK_DISABLE If set to 1, a split lock will cause an #AC(0) exception. See Section 10.1.2.3, "Features to Disable Bus Locks."	
63:30	Reserved.	
Register Address: 7AH, 122	IA32_FEATURE_ACTIVATION	
Feature Activation (R/W) Implements Feature Activation command. WRMSR to this address activates all 'activatable' features on this thread. See Table 2-2.		
Register Address: 80H, 128	MSR_TRACE_HUB_STH_ACPIBAR_BASE	
MSR_TRACE_HUB_STH_ACPIBAR_BASE (R/W) This register is used by BIOS to program Trace Hub STH base address that will be used by AET messages.		Thread
0	LOCK Lock bit. If set, this MSR cannot be re-written anymore. The lock bit has to be set in order for the AET packets to be directed to Trace Hub MMIO.	
17:1	Reserved.	
45:18	ADDRESS AET target address in Trace Hub MMIO space.	
63:46	Reserved.	
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration (R/W)		Core
3:0	PKG_C_STATE_LIMIT Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings may be supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10	
7:4	MAX_CORE_C_STATE Possible values are: 0000—reserved; 0001—C1; 0010—C3, 0011—C6.	
9:8	Reserved.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
10	IO_MWAIT_REDIRECTION_ENABLE When set, will map IO_read instructions sent to IO registers PMG_IO_BASE_ADDR.PMB0+0/1/2 to MWAIT(C2,3,4) instructions; applies to deepc4 too.	
14:11	Reserved.	
15	CFG_LOCK When set, locks bits 15:0 of this register for further writes, until the next reset occurs.	
24:16	Reserved.	
25	C3_STATE_AUTO_DEMOTION_ENABLE When set, processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.	
26	C1_STATE_AUTO_DEMOTION_ENABLE When set, processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.	
27	ENABLE_C3_UNDEMOTION Enable Un-Demotion from Demoted C3.	
28	ENABLE_C1_UNDEMOTION Enable Un-Demotion from Demoted C1.	
29	ENABLE_PKGC_AUTODEMOTION Enable Package C-State Auto-Demotion. It enables use of the history of past package C-state depth and residence, as a factor in determining C-State depth.	
30	ENABLE_PKGC_UNDEMOTION Enable Package C-State Un-Demotion. It enables considering cases where demotion was the incorrect decision in determining C-State depth.	
31	TIMED_MWAIT_ENABLE When set, enables Timed MWAIT feature. MWAIT would #GP on attempts to do setup MWAIT timer if this bit is not set.	
63:32	Reserved.	
Register Address: E4H, 228	MSR_IO_CAPTURE_BASE	
IO Capture Base (R/W) Power Management IO Redirection in C-state. See http://biosbits.org.		Core
15:0	LVL_2_BASE_ADDRESS Specifies the base address visible to software for IO redirection. If MSR_PKG_CST_CONFIG_CONTROL.IO_MWAIT_REDIRECTION_ENABLE, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
18:16	CST_RANGE Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL.IO_MWAIT_REDIRECTION_ENABLE: 000b—C3 is the max C-State to include. 001b—C6 is the max C-State to include. 010b—C7 is the max C-State to include.	
63:19	Reserved.	
Register Address: 13CH, 316	MSR_FEATURE_CONFIG	
AES Feature Configuration (R/W)		Core
0	AESNI_LOCK Once this bit is set, writes to this register will not be allowed.	
1	AESNI_DISABLE This bit disables Advanced Encryption Standard feature on this processor core. To disable AES, BIOS will write '11 to this MSR on every core.	
63:2	Reserved.	
Register Address: 140H, 320	MSR_FEATURE_ENABLES	
Feature Enable (R/W) Miscellaneous enables for thread specific features.		Thread
0	CPUID_GP_ON_CPL_GT_0 Causes CPUID to #GP if CPL greater than 0 and not in SMM.	
63:1	Reserved.	
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target (R/W) Legacy register holding temperature related constants for Platform use.		Package
6:0	TCC Offset Time Window Describes the RATL averaging time window.	
7	TCC Offset Clamping Bit When enabled will allow RATL throttling below P1.	
15:8	Temperature Control Offset Fan Temperature Target Offset (a.k.a. T-Control) indicates the relative offset from the Thermal Monitor Trip Temperature at which fans should be engaged.	
23:16	TCC Activation Temperature The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
30:24	TCC Activation Offset Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only if MSR_PLATFORM_INFO[30] is set.	
31	LOCKED When set, this entire register becomes read-only.	
63:2	Reserved.	
Register Address: 1A4H, 420	MSR_PREFETCH_CONTROL	
PREFETCH Control (R/W) Prefetch disable bits.		Thread
0	L2_HARDWARE_PREFETCHER_DISABLE If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.	
1	L2_ADJACENT_CACHE_LINE_PREFETCHER_DISABLE If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).	
2	DCU_HARDWARE_PREFETCHER_DISABLE If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.	
3	DCU_IP_PREFETCHER_DISABLE If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.	
4	DCU_NEXT_PAGE_PREFETCH_DISABLE If 1, disables Next Page prefetcher.	
5	AMP_PREFETCH_DISABLE If 1, disables L2 Adaptive Multipath Probability (AMP) prefetcher.	
6	LLC_PAGE_PREFETCH_DISABLE If 1, disables the LLC Page prefetcher.	
7	AOP_PREFETCH_DISABLE	
8	STREAM_PREFETCH_CODE_FETCH_DISABLE	
63:9	Reserved.	
Register Address: 1A6H, 422	MSR_OFFCORE_RSP_0	
OFFCORE_RSP_0 (R/W) Offcore Response Event Select Register		Thread
0	TRUE_DEMAND_CACHE_LOAD Demand Data Rd = DCU reads (includes partials) that is not tagged homeless.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
1	DEMAND_RFO Demand Instruction fetch = IFU Fetches. ItoM or RFO that is not tagged homeless.	
2	DEMAND_CODE_READ Demand Instruction fetch = IFU Fetches. CRd or CRd_UC.	
3	CORE_MODIFIED_WRITEBACK WBMtoI or WBMtoE.	
4	HW_PREFETCH_MLC_LOAD L2 prefetcher requests triggered by reads from MEC (except those triggered by I-side).	
5	HW_PREFETCH_MLC_RFO L2 prefetcher requests triggered by RFOs.	
6	HW_PREFETCH_MLC_CODE L2 prefetcher requests triggered by I-side requests.	
7	HW_PREFETCH_LLC_LOAD LLC prefetch requests triggered by DRd.	
8	HW_PREFETCH_LLC_RFO LLC prefetch requests triggered by RFO.	
9	HW_PREFETCH_LLC_CODE LLC prefetch requests triggered by CRd.	
10	L1_HWPREFETCH Covers Hardware PFRFO, PFNEAR, PFMED, PFFAR, PFHW, PFNTA, PFNPP, PFIPP including the homeless versions.	
11	ALL_STREAMING_STORE Write Combining. WCiL or WCiLF.	
12	CORE_NON_MODIFIED_WB WBEFtoI or WBEFtoE.	
13	LLC_PREFETCH LLC prefetch of load/code/RFO.	
14	L1_SWPREFETCH Covers Software PFRFO, PFNEAR, PFMED, PFFAR, PFHW, PFNTA, PFNPP, PFIPP including the homeless versions.	
15	OTHER Includes CLFlush, CLFlushOPT, CLDemote, CLWB, Enqueue SetMonitor, PortIn, IntA, Lock, SplitLock, Unlock, SpCyc, ClrMonitor, PortOut, IntPriUp, IntLog, IntPhy, EOI, RdCurr, WbStoI, LLCWBInv, LLCInv, NOP, PCOMMIT.	
16	ANY_RESP Match on any response.	
17	SUPPLIER_NONE No Supplier Details. DATA_PRE [6:3] = 0.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
18	LLC_HIT_M_STATE LLC/L3, M-state, DATA_PRE [6:3] = 2.	
19	LLC_HIT_E_STATE LLC/L3, E-state, DATA_PRE [6:3] = 4.	
20	LLC_HIT_S_STATE LLC/L3, S-state, DATA_PRE [6:3] = 6.	
21	LLC_HIT_F_STATE LLC/L3, F-state, DATA_PRE [6:3] = 8.	
22	FAR_MEM_LOCAL Far Memory, Local, DATA_PRE [6:3] = 1.	
23	FAR_MEM_REMOTE_0_HOP Far Memory, Remote 0-hop, DATA_PRE [6:3] = 3.	
24	FAR_MEM_REMOTE_1_HOP Far Memory, Remote 1-hop, DATA_PRE [6:3] = 5.	
25	FAR_MEM_REMOTE_2_PLUS_HOP Far Memory, Rem 2+ hop, DATA_PRE [6:3] = 7.	
26	NEAR_MEM_MISS_LOCAL_NODE LLC Miss Local Node. Near Memory, Local DATA_PRE [6:3] = E.	
27	NEAR_MEM_REMOTE_0_HOP Near Memory, Remote 0-hop, DATA_PRE [6:3] = B	
28	NEAR_MEM_REMOTE_1_HOP Near Memory, Remote 1-hop, DATA_PRE [6:3] = D.	
29	NEAR_MEM_REMOTE_2_PLUS_HOP Near Memory, Remote 2+ hop, DATA_PRE [6:3] = F.	
30	SPL_HIT Snoop Info: SPL-hit, DATA_PRE [2:0] = 6.	
31	SNOOP_NONE No details as to Snoop-related info. Snoop Info: None, DATA_PRE [2:0] = 0.	
32	NOT_NEEDED No snoop was needed to satisfy the request. Snoop Info: Not needed, DATA_PRE [2:0] = 1.	
33	MISS No snoop was needed to satisfy the request. Snoop Info: Miss, DATA_PRE [2:0] = 2.	
34	HIT_NO_FWD A snoop was needed and it Hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. Snoop Info: Hit No Fwd, DATA_PRE [2:0] = 3.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
35	HIT_EF_WITH_FWD A snoop was needed and data was Forwarded from a remote socket. Snoop Info: Hit EF w/Fwd, DATA_PRE [2:0] = 4.	
36	HITM A snoop was needed and it HitMed in local or remote cache. HitM denotes a cache-line was modified before snoop effect. Snoop Info: HitM, DATA_PRE [2:0] = 5.	
37	NON_DRAM Target was non-DRAM system address. Snoop Info: HitM, DATA_PRE [2:0] = 5.	
38	GO_ERR GO-ERR, RspData[3:0] = 0100.	
39	GO_NO_GO GO-NoGO, RspData[3:0] = 0111.	
40	INPKG_MEM_LOCAL In-package Memory, Local, DATA_PRE [6:3] = 9.	
41	INPKG_MEM_NONLOCAL In-package Memory, Non-Local, DATA_PRE [6:3] = C.	
43:42	Reserved.	
44	UC_LOAD PRd or UCRdF.	
45	UC_STORE WiL.	
46	PARTIAL_STREAMING_STORES WCiL.	
47	FULL_STREAMING_STORES WCiLF.	
48	L1_MODIFIED_WB EVICTION EXTTYPE from MEC.	
49	L2_MODIFIED_WB WBMtoI or WBMtoE.	
50	PSMI MemPushWr_NS (PSMI only).	
51	ITOM ItoM.	
63:52	Reserved.	
Register Address: 1A7H, 423	MSR_OFFCORE_RSP_1	
OFFCORE_RSP_1 (R/W) Offcore Response Event Select Register. See MSR_OFFCORE_RSP_0 (at1A6H).		Thread
Register Address: 1AAH, 426	MSR_MISC_PWR_MGMT	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Miscellaneous Power Management Control (R/W) Various model-specific features enumeration. See http://biosbits.org.		Package
0	Reserved.	
1	ENABLE_HWP_VOTING_RIGHT When set (1), The CPU will take into account thread HWP requests for threads that have voting rights only (ignores thread requests if they do not have voting rights). When reset(0), The CPU will take into account all thread HWP requests, even for threads that don't have voting rights. Setting this bit will cause the HWP Base feature bit to be reported in CPUID as present; clearing will cause it to be reported as non-present.	
5:2	Reserved.	
6	ENABLE_HWP Setting this bit will cause the HWP Base feature bit to report as present in CPUID; clearing this bit will cause CPUID to report the feature as non-present.	
7	ENABLE_HWP_INTERRUPT Setting this bit will cause the HWP Interrupt feature CPUID[6].EAX[8] bit to report as present; clearing will report as non-present.	
8	ENABLE_OUT_OF_BAND_AUTONOMOUS Setting this bit will cause the HWP Autonomous feature bit to report as present; clearing will report as non-present.	
11:9	Reserved.	
12	ENABLE_HWP_EPP Enable HWP EPP. Setting this bit (1) will cause the HWP CPUID[6].EAX[10] Energy Performance Preference bit to report as present (1); clearing will report as non-present (0).	
13	LOCK Setting this bit will prevent the BIOS specific bits from changing until the next reset. i.e., only Bits [0,22] which are meant for OS use can be changed once the LOCK bit is set.	
63:14	Reserved.	
Register Address: 1ADH, 429	MSR_PRIMARY_TURBO_RATIO_LIMIT	
Primary Maximum Turbo Ratio Limit (R/W) Software can configure these limits when MSR_PLATFORM_INFO[28] = 1. Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.		Package
7:0	MAX_TURBO_GROUP_0: Maximum turbo ratio limit with 1 core active.	
15:8	MAX_TURBO_GROUP_1: Maximum turbo ratio limit with 2 cores active.	
23:16	MAX_TURBO_GROUP_2: Maximum turbo ratio limit with 3 cores active.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:24	MAX_TURBO_GROUP_3: Maximum turbo ratio limit with 4 cores active.	
39:32	MAX_TURBO_GROUP_4: Maximum turbo ratio limit with 5 cores active.	
47:40	MAX_TURBO_GROUP_5: Maximum turbo ratio limit with 6 cores active.	
55:48	MAX_TURBO_GROUP_6: Maximum turbo ratio limit with 7 cores active.	
63:56	MAX_TURBO_GROUP_7: Maximum turbo ratio limit with 8 cores active.	
Register Address: 1F1H, 497	MSR_CRASHLOG_CONTROL	
Crash Log Control (R/W) Write data to a Crash Log configuration.		Thread
0	CDDIS CrashDump_Disable: If set, indicates that Crash Dump is disabled.	
1	EN_GPRS Collect GPRs on a crash dump. Only meaningful when CDDIS is zero.	
2	EN_GPRS_IN_SMM Collect GPRs in SMM on a crash dump. Only meaningful when CDDIS is zero. EN_GPRS will override this control,	
3	TRIPLE_FAULT_SHUTDOWN Collect a crash log on a triple fault shutdown. Only meaningful when CDDIS is zero.	
63:4	Reserved.	
Register Address: 1F5H, 501	MSR_PRMRR_PHYS_MASK	
Processor Reserved Memory Range Register - Physical Mask (R/W)		Core
9:0	Reserved.	
10	LOCK Once set, this bit prevents software from modifying the PRMRR.	
11	VALID This bit serves as the enable for the PRMRR; the PRMRR must be LOCKed before it can be enabled.	
19:12	Reserved.	
45:20	MASK PRMRR Address Mask.	
63:46	Reserved.	
Register Address: 1FCH, 508	MSR_POWER_CTL	
Power Control Register (R/W) See http://biosbits.org.		Package

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
0	ENABLE_BIDIR_PROCHOT Used to enable or disable the response to PROCHOT# input. When set/enabled, the platform can force the CPU to throttle to a lower power condition such as Pn/Pm by asserting prochot#. When clear/disabled (default), the CPU ignores the status of the prochot input signal.	
1	C1E_ENABLE When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).	
2	SAPM_IMC_C2_POLICY This bit determines if self-refresh activation is allowed when entering Package C2 State. If it is set to 0b, PCODE will keep the FORCE_SR_OFF bit asserted in Package C2 State and allow its negation according to the defined latency negotiations with the PCH and Display Engine in Package C3 and deeper states. Otherwise, self-refresh is allowed in Package C2 State.	
3	FAST_BRK_SNP_EN This bit controls the VID swing rate for the OTHER_SNP_WAKE events that are detected by the iMPH. This is the event that is detected by the iMPH when a non-DMI snoopable request is observed while UCLK domain is not functional. 0b: Use slow VID swing rate. 1b: Use fast VID swing rate.	
17:4	Reserved.	
18	PWR_PERF_PLTFRM_OVR Power performance platform override.	
19	EE_TURBO_DISABLE Setting this bit disables the P-States energy efficiency optimization. Default value is 0. Disable/enable the energy efficiency optimization in P-State legacy mode (when IA32_PM_ENABLE[HWP_ENABLE] = 0), has an effect only in the turbo range or into PERF_MIN_CTL value if it is not zero set. In HWP mode (IA32_PM_ENABLE[HWP_ENABLE] == 1), has an effect between the OS desired or OS maximize to the OS minimize performance setting.	
20	RTH_DISABLE Setting this bit disables the Race to Halt optimization and avoids this optimization limitation to execute below the most efficient frequency ratio. Default value is 0 for processors that support Race to Halt optimization.	
21	DIS_PROCHOT_OUT Prochot output disable.	
22	PROCHOT_RESPONSE Prochhot configurable response enable.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
23	VR_THERM_ALERT_DISABLE_LOCK When set to 1, locks PROCHOT related bits of this MSR. Once set, a reset is required to clear this bit.	
24	VR_THERM_ALERT_DISABLE When set to 1, disables the VR_THERMAL_ALERT signaling.	
25	DISABLE_RING_EE Disable Ring EE.	
26	DISABLE_SA_OPTIMIZATION Disable SA optimization.	
27	DISABLE_OOK Disable OOK.	
28	DISABLE_AUTONOMOUS Disable HWP autonomous mode.	
29	Reserved.	
30	CSTATE_PREWAKE_DISABLE C-state pre-wake disable.	
63:31	Reserved.	
Register Address: 2A0H, 672	MSR_PRMRR_BASE_0	
Processor Reserved Memory Range Register - Physical Base Control Register (R/W)		Core
2:0	MEMTYPE Memory type for PRMRR accesses.	
3	CONFIGURED PRMRR base configured.	
19:4	Reserved.	
45:20	BASE PRMRR base address.	
63:46	Reserved.	
Register Address: 474H, 1140	IA32_MC29_CTL	
MC29_CTL. See Table 2-2.		Package
Register Address: 475H, 1141	IA32_MC29_STATUS	
MC29_STATUS. See Table 2-2.		Package
Register Address: 476H, 1142	IA32_MC29_ADDR	
MC29_ADDR. See Table 2-2.		Package
Register Address: 477H, 1143	IA32_MC29_MISC	
MC29_MISC. See Table 2-2.		Package
Register Address: 478H, 1144	IA32_MC30_CTL	
MC30_CTL. See Table 2-2.		Package
Register Address: 479H, 1145	IA32_MC30_STATUS	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MC30_STATUS. See Table 2-2.		Package
Register Address: 47AH, 1146	IA32_MC30_ADDR	
MC30_ADDR. See Table 2-2.		Package
Register Address: 47BH, 1147	IA32_MC30_MISC	
MC30_MISC. See Table 2-2.		Package
Register Address: 47CH, 1148	IA32_MC31_CTL	
MC31_CTL. See Table 2-2.		Package
Register Address: 47DH, 1149	IA32_MC31_STATUS	
MC31_STATUS. See Table 2-2.		Package
Register Address: 47EH, 1150	IA32_MC31_ADDR	
MC31_ADDR. See Table 2-2.		Package
Register Address: 47FH, 1151	IA32_MC31_MISC	
MC31_MISC. See Table 2-2.		Package
Register Address: 4E0H, 1248	MSR_SMM_FEATURE_CONTROL	
Enhanced SMM Feature Control (R/W) Reports SMM capability enhancement.		Package
0	LOCK When set, locks this register from further changes.	
1	SMM_CPU_SAVE_EN If 0, SMI/RSM will save/restore state in SMRAM If 1, SMI/RSM will save/restore state from SRAM.	
2	SMM_CODE_CHK_EN When clear (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set, any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.	
63:3	Reserved.	
Register Address: 601H, 1537	MSR_VR_CURRENT_CONFIG	
Power Limit 4 (PL4) (R/W) Package-level maximum power limit (in Watts). It is a proactive, instantaneous limit.		Package
15:0	CURRENT_LIMIT PL4 Value in 0.125 A increments. This field is locked by MSR_VR_CURRENT_CONFIG.LOCK. When the LOCK bit is set to 1, this field becomes Read Only.	
30:16	Reserved.	
31	LOCK This bit will lock the CURRENT_LIMIT settings in this register and will also lock this setting. This means that once set to 1, the CURRENT_LIMIT setting and this bit become Read Only until the next Warm Reset.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:32	Reserved.	
Register Address: 620H, 1568	MSR_UNCORE_RATIO_LIMIT	
Uncore Ratio Limit (R/W) Min/Max Ratio Limits for Uncore LLC and Ring.		Package
6:0	MAX_CLR_RATIO Maximum allowed ratio for the Ring and Last Level Cache (LLC).	
7	Reserved.	
14:8	MIN_CLR_RATIO Minimum allowed ratio for the Ring and Last Level Cache (LLC).	
63:15	Reserved.	
Register Address: 638H, 1592	MSR_PP0_POWER_LIMIT	
MSR_PP0_POWER_LIMIT (R/W) PP0 RAPL power unit control.		Package
14:0	IA_PP_PWR_LIM This is the power limitation on the IA cores power plane. The unit of measurement is defined in PACKAGE_POWER_SKU_UNIT_MSR[PWR_UNIT].	
15	PWR_LIM_CTRL_EN This bit must be set in order to limit the power of the IA cores power plane. 0b: IA cores power plane power limitation is disabled. 1b: IA cores power plane power limitation is enabled.	
16	PP_CLAMP_LIM Power Plane Clamping limitation; allow going below P1. 0b: PBM is limited between P1 and P0. 1b: PBM can go below P1.	
23:17	CTRL_TIME_WIN x = CTRL_TIME_WIN[23:22] y = CTRL_TIME_WIN[21:17] The timing interval window is Floating Point number given by 1.x * power(2,y). The unit of measurement is defined in PACKAGE_POWER_SKU_UNIT_MSR[TIME_UNIT]. The maximal time window is bounded by PACKAGE_POWER_SKU_MSR[PKG_MAX_WIN]. The minimum time window is 1 unit of measurement (as defined above).	
30:24	Reserved.	
31	PP_PWR_LIM_LOCK When set, all settings in this register are locked and are treated as Read Only.	
63:32	Reserved.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 64FH, 1615	MSR_CORE_PERF_LIMIT_REASONS	
Core Performance Limit Reasons Indicator of Frequency Clipping in Processor Cores. (Frequency refers to processor core frequency.)		Package
0	PROCHOT (R/O) PROCHOT Status. When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.	
1	THERMAL (R/O) Thermal Status. When set, frequency is reduced below the operating system request due to a thermal event.	
3:2	Reserved.	
4	RSR_LIMIT (R/O) Residency State Regulation Status. When set, frequency is reduced below the operating system request due to residency state regulation limit.	
5	RATL (R/O) Running Average Thermal Limit Status. When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).	
6	VR_THERMALERT (R/O) VR Therm Alert Status. When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).	
7	VR_TDC (R/O) VR Therm Design Current Status. When set, frequency is reduced below the operating system request due to VR thermal design current limit.	
8	OTHER (R/O) Other Status. When set, frequency is reduced below the operating system request due to electrical or other constraints.	
9	Reserved.	
10	PBM_PL1 (R/O) Package/Platform-Level Power Limiting PL1 Status. When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.	
11	PBM_PL2 (R/O) Package/Platform-Level PL2 Power Limiting Status. When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.	
12	MAX_TURBO_LIMIT (R/O) Max Turbo Limit Status. When set, frequency is reduced below the operating system request due to multi-core turbo limits.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
13	TURBO_ATTEN (R/O) Turbo Transition Attenuation Status. When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.	
15:14	Reserved.	
16	PROCHOT_LOG (R/W) PROCHOT Log. When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	THERMAL_LOG (R/W) Thermal Log When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
19:18	Reserved.	
20	RSR_LIMIT_LOG (R/W) Residency State Regulation Log. When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
21	RATL_LOG (R/W) Running average thermal limit Log, RW, When set by PCODE indicates that Running average thermal limit has cause IA frequency clipping. Software should write to this bit to clear the status in this bit.	
22	VR_THERMALERT_LOG (R/W) VR Therm Alert Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	VR_TDC_LOG (R/W) VR Thermal Design Current Log. When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
24	OTHER_LOG (R/W) Other Log. When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Reserved.	
26	PBM_PL1_LOG (R/W) Package/Platform-Level PL1 Power Limiting Log. When set, indicates that the Package or Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
27	PBM_PL2_LOG (R/W) Package/Platform-Level PL2 Power Limiting Log. When set, indicates that the Package or Platform Level PL2/PL3 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	MAX_TURBO_LIMIT_LOG (R/W) Max Turbo Limit Log. When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
29	TURBO_ATTEN_LOG (R/W) Turbo Transition Attenuation Log. When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:30	Reserved.	
Register Address: 650H, 1616	MSR_SECONDARY_TURBO_RATIO_LIMIT	
Secondary Maximum Turbo Ratio Limit (R/W) Software can configure these limits when MSR_PLATFORM_INFO[28] = 1. Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.		Package
7:0	MAX_TURBO_GROUP_0: Maximum turbo ratio limit with 1 core active.	
15:8	MAX_TURBO_GROUP_1: Maximum turbo ratio limit with 2 cores active.	
23:16	MAX_TURBO_GROUP_2: Maximum turbo ratio limit with 3 cores active.	
31:24	MAX_TURBO_GROUP_3: Maximum turbo ratio limit with 4 cores active.	
39:32	MAX_TURBO_GROUP_4: Maximum turbo ratio limit with 5 cores active.	
47:40	MAX_TURBO_GROUP_5: Maximum turbo ratio limit with 6 cores active.	
55:48	MAX_TURBO_GROUP_6: Maximum turbo ratio limit with 7 cores active.	
63:56	MAX_TURBO_GROUP_7: Maximum turbo ratio limit with 8 cores active.	
Register Address: 65CH, 1628	MSR_PLATFORM_POWER_LIMIT	
Platform Power Limit Control (R/W) Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.		Package

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
14:0	POWER_LIMIT_1 Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field. The default value is the Thermal Design Power (a.k.a TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT.	
15	POWER_LIMIT_1_EN When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit 1 over the time window specified by Power Limit 1 Time Window.	
16	CRITICAL_POWER_CLAMP_1 When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit 1 value.	
23:17	POWER_LIMIT_1_TIME Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation: Time Window = (float) ((1+(X/4))*(2^Y)), where: X = POWER_LIMIT_1_TIME[23:22] Y = POWER_LIMIT_1_TIME[21:17] The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is 0DH, The unit is specified in MSR_RAPLPOWER_UNIT[Time Unit]	
31:24	Reserved.	
46:32	POWER_LIMIT_2 Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor. The recommended default value is 1.25 times the Long Duration Power Limit (i.e., Platform Power Limit 1).	
47	POWER_LIMIT_2_EN When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit 2 over the Short Duration time window.	
48	CRITICAL_POWER_CLAMP_2 When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit 2 value.	
62:49	Reserved.	
63	LOCK Setting this bit will lock all other bits of this MSR until system RESET.	
Register Address: 6B0H, 1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MSR_GRAPHICS_PERF_LIMIT_REASONS Indicator of Frequency Clipping in the Processor Graphics. (Frequency refers to processor graphics frequency.)		Package
0	PROCHOT (R/O) PROCHOT Status. When set, frequency is reduced due to assertion of external PROCHOT.	
1	THERMAL (R/O) Thermal Status. When set, frequency is reduced due to a thermal event.	
4:2	Reserved.	
5	RATL (R/O) Running Average Thermal Limit Status. When set, frequency is reduced due to running average thermal limit.	
6	VR_THERMALERT (R/O) VR Therm Alert Status. When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.	
7	VR_TDC (R/O) VR Thermal Design Current Status. When set, frequency is reduced due to VR TDC limit.	
8	OTHER (R/O) Other Status. When set, frequency is reduced due to electrical or other constraints.	
9	Reserved.	
10	PBM_PL1 (R/O) Package/Platform-Level Power Limiting PL1 Status. When set, frequency is reduced due to package/platform-level power limiting PL1.	
11	PBM_PL2 (R/O) Package/Platform-Level PL2 Power Limiting Status. When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.	
12	INEFFICIENT_OPERATION (R/O) Inefficient Operation Status. When set, processor graphics frequency is operating below target frequency.	
15:13	Reserved.	
16	PROCHOT_LOG (R/W) PROCHOT Log. When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	THERMAL_LOG (R/W) Thermal Log. When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
20:18	Reserved.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
21	RATL_LOG (R/W) Running Average Thermal Limit Log. When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR_THERMALERT_LOG (R/W) VR Therm Alert Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	VR_TDC_LOG (R/W) VR Thermal Design Current Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
24	OTHER_LOG (R/W) Other Log. When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Reserved.	
26	PBM_PL1_LOG (R/W) Package/Platform-Level PL1 Power Limiting Log. When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
27	PBM_PL2_LOG (R/W) Package/Platform-Level PL2 Power Limiting Log. When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
28	INEFFICIENT_OPERATION_LOG (R/W) Inefficient Operation Log. When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:29	Reserved.	
Register Address: 6B1H, 1713	MSR_RING_PERF_LIMIT_REASONS	
MSR_RING_PERF_LIMIT_REASONS Indicator of Frequency Clipping in the Ring Interconnect. (Frequency refers to ring interconnect in the uncore.)		Package
0	PROCHOT (R/O) PROCHOT Status. When set, frequency is reduced due to assertion of external PROCHOT.	
1	THERMAL (R/O) Thermal Status. When set, frequency is reduced due to a thermal event.	
4:2	Reserved.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
5	RATL (R/O) Running Average Thermal Limit Status. When set, frequency is reduced due to running average thermal limit.	
6	VR_THERMALERT (R/O) VR Therm Alert Status. When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.	
7	VR_TDC (R/O) VR Thermal Design Current Status. When set, frequency is reduced due to VR TDC limit.	
8	OTHER (R/O) Other Status. When set, frequency is reduced due to electrical or other constraints.	
9	Reserved.	
10	PBM_PL1 (R/O) Package/Platform-Level Power Limiting PL1 Status. When set, frequency is reduced due to package/platform-level power limiting PL1.	
11	PBM_PL2 (R/O) Package/Platform-Level PL2 Power Limiting Status. When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.	
15:12	Reserved.	
16	PROCHOT_LOG (R/W) PROCHOT Log. When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
17	THERMAL_LOG (R/W) Thermal Log. When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
20:18	Reserved.	
21	RATL_LOG (R/W) Running Average Thermal Limit Log. When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
22	VR_THERMALERT_LOG (R/W) VR Therm Alert Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
23	VR_TDC_LOG (R/W) VR Thermal Design Current Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
24	OTHER_LOG (R/W) Other Log. When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
25	Reserved.	
26	PBM_PL1_LOG (R/W) Package/Platform-Level PL1 Power Limiting Log. When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
27	PBM_PL2_LOG (R/W) Package/Platform-Level PL2 Power Limiting Log. When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.	
63:28	Reserved.	
Register Address: 9FBH, 2555	IA32_TME_CLEAR_SAVED_KEY	
IA32_TME_CLEAR_SAVED_KEY (R/W) See Table 2-2.		Package
Register Address: 9FFH, 2559	MSR_CORE_MKTME_ACTIVATE	
MSR_CORE_MKTME_ACTIVATE (R/O) MSR to read TME_ACTIVATE[MK_TME_KEYID_BITS].		Core
31:0	Reserved.	
35:32	READ_MK_TME_KEYID_BITS This value will be returned on a RDMSR, but must be zero on a WRMSR.	
63:36	Reserved.	

The MSRs listed in Table 2-54 are unique to the Intel Core Ultra 7 processor P-core. These MSRs are not supported on the processor E-core.

Table 2-54. MSRs Supported by the Intel® Core™ Ultra 7 Processor P-core

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 30CH, 780	IA32_FIXED_CTR3	
Fixed-Function Performance Counter 3 (R/W)		Thread
47:0	FIXED_COUNTER Top-down Microarchitecture Analysis unhalted number of available slots counter.	
63:48	Reserved.	
Register Address: 329H, 809	MSR_PERF_METRICS	

Table 2-54. MSRs Supported by the Intel® Core™ Ultra 7 Processor P-core (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Performance Metrics (R/W) This register provides built-in support for Top-down Micro-architecture Analysis (TMA) metrics. It exposes the four TMA Level 1 metrics where the lower 32 bits are divided into four 8 bit fields, each of which is an integer percentage of the total TOPDOWN.SLOTS (as reported by fixed counter 3).		Thread
7:0	RETIRING Percent of utilized by uops that eventually retire (commit).	
15:8	BAD_SPECULATION Percent of Wasted due to incorrect speculation, covering Utilized by uops that do not retire, or Recovery Bubbles (unutilized slots).	
23:16	FRONTEND_BOUND Percent of Unutilized slots where Front-end did not deliver a uop while Back-end is ready.	
31:24	BACKEND_BOUND Percent of Unutilized slots where a uop was not delivered to Back-end due to lack of Back-end resources.	
39:32	MULTI_UOPS Frontend bound.	
47:40	BRANCH_MISPREDICTS Frontend bound.	
55:48	FRONTEND_LATENCY Frontend bound.	
63:56	MEMORY_BOUND Frontend bound.	
Register Address: 540H, 1344	MSR_THREAD_UARCH_CTL	
Thread Microarchitectural Control (R/W) See Table 2-47.		Thread
Register Address: 541H, 1345	MSR_CORE_UARCH_CTL	
Core Microarchitecture Control MSR (R/W) See Table 2-44.		Core

The MSRs listed in Table 2-48 are unique to the Intel Core Ultra 7 processor E-core. These MSRs are not supported on the processor P-core.

Table 2-55. MSRs Supported by the Intel® Core™ Ultra 7 Processor E-core

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 4F0H, 1264	MSR_SAF_CTRL	
SAF Control (W/O) Extension to SAF.		Package
0	INVALIDATE_CURRENT_STRIDE Invalidate all chunks in current stride.	
63:1	Reserved.	

Table 2-55. MSRs Supported by the Intel® Core™ Ultra 7 Processor E-core (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: D18H—D1FH, 3352—3359	IA32_L2_MASK_[8-15]	
IA32_L2_MASK_[8-15] (R/W) If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] ≥ 0. Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 19, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features."		Module
15:0	WAY_MASK Capacity Bit Mask. Available ways vectors for class of service of IA core. '1 in bit indicates allocation to the way is allowed. '0 indicates allocation to the way is not allowed.	
31:16	Reserved.	
Register Address: 1309H—130BH, 4873—4875	MSR_RELOAD_FIXED_CTRx	
Reload value for IA32_FIXED_CTRx (R/W)		Thread
47:0	Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and FIXED_CTRx is overflowed.	
63:48	Reserved.	
Register Address: 14C1H—14C8H, 5313 —5320	MSR_RELOAD_PMCx	
Reload value for IA32_PMCx (R/W)		Thread
47:0	Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and PMCx is overflowed.	
63:48	Reserved.	
Register Address: 1A8EH, 6798	MSR_STLB_FILL_TRANSLATION	
STLB Fill Translation (W/O) STLB QoS MSR to fill translations into STLB.		Core
3:0	CLOS Class of service to use for the fill.	
9:4	Reserved.	
10	X Set to 1 when LA is to an executable page.	
11	RW Set to 1 when LA is to a writeable page.	
63:12	LA Logical address to use for fill.	

2.17.10 MSRs Introduced in the Intel® Xeon® 6 P-Core Processors

Table 2-56 lists additional MSRs for the Intel Xeon 6 P-core processors. Intel Xeon 6 P-core processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_ADH or 06_AEH.

For an MSR listed in Table 2-56 that also appears in the model-specific tables of prior generations, Table 2-56 supersedes prior generation tables.

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 33H, 51	MSR_MEMORY_CONTROL	
MSR_MEMORY_CONTROL (R/W) Disables split locks, which are locked instructions that split a cache line.		Core
26:0	Reserved.	
27	UC_STORE_THROTTLE If set to 1, when enabled, the processor allows one in-progress, post-retirement UC stores at a time.	
28	UC_LOCK_DISABLE If set to 1, a UC load lock will trigger a fault. If clear to 0, UC load locks proceed normally.	
29	SPLIT_LOCK_DISABLE If set to 1, a split lock will trigger an #AC fault. If clear to 0, split locks proceed normally	
63:30	Reserved.	
Register Address: 34H, 52	MSR_SMI_COUNT	
SMI Counter (R/W)		Thread
31:0	SMI_COUNT Running count of SMI events since the last reset.	
63:32	Reserved.	
Register Address: 39H, 57	MSR_SOCKET_ID	
Socket ID (R/W) Reassigns the package-specific portions of the APIC ID. This MSR is used on scalable DP and high-end MP platforms to resolve legacy-mode APIC ID conflicts.		Package
10:0	PACKAGE_ID: Holds package ID. This reflects the upper bits of the APIC ID.	
63:11	Reserved.	
Register Address: 7AH, 122	IA32_FEATURE_ACTIVATION	
IA32_FEATURE_ACTIVATION (R/W) Implements Feature Activation command. WRMSR to this address activates all 'activatable' features on this thread. See Table 2-2.		Thread
Register Address: 7BH, 123	IA32_MCU_ENUMERATION	
IA32_MCU_ENUMERATION (R/O) Enumeration of architectural features. See Table 2-2.		Package
Register Address: 7CH, 124	IA32_MCU_STATUS	
IA32_MCU_STATUS (R/O) Communicates results from the previous patch loads. See Table 2-2.		Package
Register Address: 82H, 130	IA32_FZM_RANGE_INDEX	
IA32_FZM_RANGE_INDEX (R/W) Index and Domain handle for a valid FZM region. Programmed by SW and used by other FRM MSRs FZM Range Index register to R/W Domain Index. See Table 2-2.		Thread

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 83H, 131	IA32_FZM_DOMAIN_CONFIG	
IA32_FZM_DOMAIN_CONFIG (R/O) Bit mask of valid regions within the domain identified by FZM_RANGE_INDEX. See Table 2-2.		Thread
Register Address: 84H, 132	IA32_FZM_RANGE_STARTADDR	
IA32_FZM_RANGE_STARTADDR (R/O) Start address of the FZM range pointed to by FZM_RANGE_INDEX. See Table 2-2.		Thread
Register Address: 85H, 133	IA32_FZM_RANGE_ENDADDR	
IA32_FZM_RANGE_ENDADDR (R/O) End address of the specified domain in FZM_RANGE_INDEX. See Table 2-2.		Thread
Register Address: 86H, 134	IA32_FZM_RANGE_WRITESTATUS	
IA32_FZM_RANGE_WRITESTATUS (R/O) Write status of the FZM range pointed to by FZM_RANGE_INDEX. See Table 2-2.		Thread
Register Address: 87H, 135	IA32_MKTME_KEYID_PARTITIONING	
MKTME KEY ID Partitioning (R/O) Enumerates the number of activated KeyIDs for Intel TME-MK and Intel TDX. See Table 2-2.		Package
Register Address: 90H, 144	IA32_SGXLEPUBKEYHASH4	
IA32_SGXLEPUBKEYHASH4 (R/W) See Table 2-2.		Thread
Register Address: 91H, 145	IA32_SGXLEPUBKEYHASH5	
IA32_SGXLEPUBKEYHASH5 (R/W) See Table 2-2.		Thread
Register Address: 98H, 152	MSR_SEAM_WBINVDP	
SEAM WBINVDP (R/W) Allows software to WBINVD sections of the LLC.		Thread
63:0	HANDLE Caches sub-block to invalidate.	
Register Address: 99H, 153	MSR_SEAM_WBNOINVDP	
SEAM WBNOINVDP (R/W) Allows software to WBNOINVD sections of the LLC.		Thread
63:0	HANDLE Caches sub-block to invalidate.	
Register Address: 9AH, 154	MSR_SEAM_INTR_PENDING	
SEAM Interrupt Pending (R/O) Report out some event pending bits.		Thread
0	INTR Interrupt is pending.	
1	NMI NMI is pending.	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
2	SMI SMI is pending.	
4:3	OTHER_EVENTS Other events pending.	
63:5	Reserved.	
Register Address: 9BH, 155	IA32_SMM_MONITOR_CTL	
SMM Monitor Control (R/W) The SMM Monitor Configuration involves SMM code specifying the MSEG location and enabling dual-monitor treatment by writing to the corresponding MSR. See Table 2-2.		Thread
Register Address: CFH, 207	IA32_CORE_CAPABILITIES	
IA32 Core Capabilities Register (R/W) If CPUID.(EAX=07H, ECX=0):EDX[30] = 1. This MSR provides an architectural enumeration function for model-specific behavior.		Core
0	STLB_QOS When set to 1, processor supports STLB QoS.	
1	Reserved.	
2	INTEGRITY_SUPPORTED When set to 1, processor supports Functional Safety. Specific FUSA capabilities are enumerated in MSR_FUSA_CAPABILITIES.	
3	RSM_IN_CPL0_ONLY Intel System Resources Defense: When set to 1, RSM will only be allowed in CPL0 and will #GP for all non-CPL0 privilege levels.	
4	UC_LOCK_DISABLE When set to 1, processor supports UC load lock disable.	
5	SPLIT_LOCK_DISABLE When set to 1, processor supports #AC on split locks.	
6	SNP_FILTER_QOS When set to 1, processor supports Snoop Filter Quality of Service MSRs.	
7	UC_STORE_THROTTLING When set to 1, processor supports UC store throttling through MSR_MEMORY_CTRL[UC_STORE_THROTTLE].	
63:8	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: FEH, 254	IA32_MTRRCAP	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Memory Type Range Register (R/O) See Table 2-2.		Core
Register Address: 105H, 261	MSR_ARRAY_BIST	
MSR_ARRAY_BIST (R/W) Triggered by writing and reading an MSR that can be written by Ring 0 software.		Core
31:0	ARRAY_LIST: Bit map which indicates which arrays to run MarchC- BIST ▪ Bit[0] MLC Data ▪ Bit[1] MLC Tag ▪ Bit[2] C6SRAM Data (NOP for WRMSR – used for reporting error only) ▪ Bit[3] PMA BIST (NOP for WRMSR – used for reporting error only) ▪ Bit[4] STLB Data ▪ Bit[5] IFU Data ▪ Bit[6] STLB Tag ▪ Bit[7] DCU Data ▪ Bit[8] DSB Data ▪ Bit[9] TMUL Data ▪ Bit[10] UROM pointer0 ▪ Bit[11] UROM pointer1-3 ▪ Bit[12] UROM pointer4-7 ▪ Bit[13] UROM unique0 ▪ Bit[14] UROM unique1/2 The WRMSR will run PBIST on all the arrays indicated in the bitmap, starting from the LSB. NOTE2: C6SRAM[Bit 2] and PMA[Bit 3] are only for reporting and do not execute BIST (done by EDX[15:0]uCode during Fusa-Reset).	
46:32	Reserved.	
62:47	Reserved.	
63	SIGNAL_MCE: Signal MCERR upon BIST failure.	
Register Address: 105H, 261	MSR_ARRAY_BIST_STATUS	
MSR_ARRAY_BIST_STATUS (R/O)		Core
31:0	ARRAY_COMPLETION _MASK Bitmap indicating which arrays from the ARRAY_BIST.ARRAY_LIST was not processed. 1 means not tested and 0 means tested.	
62:32	Reserved. Returns all 0s.	
63	PASS_FAIL: 0 means Pass on all arrays in the WRMSR(ARRAY_BIST.ARRAY_LIST) 1 means Fail on the LSB array in the RDMSR(ARRAY_BIST_STATUS.ARRAY_COMPLETION_MASK).	
Register Address: 122H, 290	IA32_TSX_CTRL	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32_TSX_CTRL (R/W) See Table 2-2.		Thread
Register Address: 140H, 320	MSR_FEATURE_ENABLES	
Miscellaneous enables for thread-specific features. (R/W)		Thread
0	AESNI_LOCK Once this bit is set, writes to this register will not be allowed.	
63:1	Reserved.	
Register Address: 1E0H, 480	IA32_LER_INFO	
IA32_LER_INFO (R/W) Last Event Record Destination IP Register. See Table 2-2.		Thread
Register Address: 1F9H, 505	IA32_CPU_DCA_CAP	
IA32_CPU_DCA_CAP (R/O) See Table 2-2.		Thread
Register Address: 2A1H, 673	MSR_PRMRR_BASE_1	
MSR_PRMRR_BASE_1 (R/W) Processor Reserved Memory Range Register - Physical Base Control Register.		Core
2:0	MEMTYPE Memory Type for PRMRR accesses.	
3	CONFIGURED PRMRR base configured.	
19:4	Reserved.	
51:20	BASE PRMRR Base address.	
63:52	Reserved.	
Register Address: 2A2H, 674	MSR_PRMRR_BASE_2	
MSR_PRMRR_BASE_2 (R/W) Processor Reserved Memory Range Register - Physical Base Control Register. See MSR_PRMRR_BASE_1 (2A1H) for reference; similar format.		Core
Register Address: 2A3H, 675	MSR_PRMRR_BASE_3	
MSR_PRMRR_BASE_3 (R/W) Processor Reserved Memory Range Register - Physical Base Control Register. See MSR_PRMRR_BASE_1 (2A1H) for reference; similar format.		Core
Register Address: 2A4H, 676	MSR_PRMRR_BASE_4	
MSR_PRMRR_BASE_4 (R/W) Processor Reserved Memory Range Register - Physical Base Control Register. See MSR_PRMRR_BASE_1 (2A1H) for reference; similar format.		Core
Register Address: 2A5H, 677	MSR_PRMRR_BASE_5	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MSR_PRMRR_BASE_5 (R/W) Processor Reserved Memory Range Register - Physical Base Control Register. See MSR_PRMRR_BASE_1 (2A1H) for reference; similar format.		Core
Register Address: 2A6H, 678	MSR_PRMRR_BASE_6	
MSR_PRMRR_BASE_6 (R/W) Processor Reserved Memory Range Register - Physical Base Control Register. See MSR_PRMRR_BASE_1 (2A1H) for reference; similar format.		Core
Register Address: 2A7H, 679	MSR_PRMRR_BASE_7	
MSR_PRMRR_BASE_7 (R/W) Processor Reserved Memory Range Register - Physical Base Control Register. See MSR_PRMRR_BASE_1 (2A1H) for reference; similar format.		Core
Register Address: 2B8H, 696	MSR_COPY_SBFT_HASHES	
MSR_COPY_SBFT_HASHES (W/O)		Module
63:0	SBFT_PROGRAM_SOURCE_ADDR EDX:EAX contains the linear address base of the SBFT Binary loaded into memory.	
Register Address: 2B9H, 697	MSR_SBFT_HASHES_STATUS	
MSR_COPY_SBFT_HASHES (R/O)		Core
15:0	CHUNK_SIZE EAX[15:0] - Chunk size of the test in KB.	
31:16	TOTAL_NUM_CHUNKS EAX[31:16] - Total number of chunks.	
39:32	ERROR_CODE - EDX[7:0] The error code refers to the LP that runs WRMSR(2B8H). ▪ 0x0: Reserved. ▪ 0x1: Attempt to copy SBFT-hashes when copy already in progress. ▪ 0x2: Secure Memory not set up correctly. ▪ 0x3: Scan-Image Header Image_info.ProgramID does not match MSR_INTEGRITY_CAPABILITIES[31:24], or scan-image header Processor-Signature doesn't match F/M/S, or scan-image header Processor-Flags doesn't match PlatformID. ▪ 0x4: Reserved. ▪ 0x5: Integrity check failed. ▪ 0x6: WRMSR(0x2B8) (ACTIVATE_SBAF) Reinstall of SBFT test image attempted when current SBFT test image is in use by other LPs. ▪ 0x7: Aborted due to #PF (Page Fault). ▪ 0x8: Unable to generate a Random Value.	
48:40	NUM_CHUNKS_IN_STRIDE EDX[16:8] - Number of Chunks in stride. This is the number of chunks that are installed. 0 in this field means that the CPU does not support strides, otherwise, stride value must be >=1.	
50:49	Reserved. EDX[18:17] - Set to all zeros.	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
62:51	MAX_CORE_LIMIT EDX[30:19] - Maximum Number of Cores that can run SBFTAFSBAF simultaneously -1. 0 means 1 core at a time.	
63	Valid. EDX[31] - Valid bit is set when COPY_SBFT_HASHES completed successfully.	
Register Address: 2BAH, 698	MSR_AUTHENTICATE_AND_COPY_SBFT_CHUNK	
MSR_AUTHENTICATE_AND_COPY_SBFT_CHUNK (W/O)		Core
63:0	BASE_CHUNK_TABLE_ADDR EDX:EAX[63:0] - Linear Address pointing to the CHUNK TABLE (TABLE_BASE).	
Register Address: 2BBH, 699	MSR_SBFT_CHUNKS_AUTHENTICATION_STATUS	
MSR_SBFT_CHUNKS_AUTHENTICATION_STATUS (R/O)		Core
15:0	NUM_VALID_CHUNKS EAX[15:0] - Total number of Valid (authenticated) chunks.	
31:16	NUM_CHUNKS_IN_STRIDE EAX[31:16] - Number of Chunks in Stride.	
39:32	ERROR_CODE EDX[7:0] ▪ 0x0 - No error reported. ▪ 0x1 - Attempt to authenticate a CHUNK already marked as authentic or is currently being installed by another core. ▪ 0x2 - CHUNK authentication error. HASH of chunk did not match expected value. ▪ 0x3 - Aborted due to #PF. ▪ 0x4 - Chunk Outside the current Stride. ▪ 0x5 - Interrupted.	
47:40	Reserved. EDX[15:8] - Set to all zeros.	
63:48	CURRENT_MAX_BUNDLE_INDX EDX[31:16] - Maximum Bundle Index in current stride.	
Register Address: 2BCH, 700	MSR_ACTIVATE_SBFT	
MSR_ACTIVATE_SBFT (W/O)		Core
13:0	SBFT_BUNDLE_INDEX EAX[13:0] - Indicates SBFT Bundle Index to start from.	
15:14	SBFT_PRGM_INDEX EAX[15:14] - Indicates what SBFT Program index to run.	
31:16	Reserved. Set to all zeros.	
62:32	THREAD_WAIT_DELAY EDX[30:0] - TSC-based delay to allow threads to rendezvous.	
63	Reserved. EDX[31] - Must be set to 0. #GP fault otherwise.	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 2BDH, 701	MSR_SBFT_STATUS	
MSR_SBFT_STATUS (R/O)		Core
13:0	SBFT_BUNDLE_INDEX EAX[13:0] - SBFT Bundle that was executed.	
15:14	SBFT_PGM_INDEX EAX[15:14] - Indicates what SBFT Program index that was last ran. Maps to same field in WRMSR(ACTIVATE_SBFT). On a test pass this field will be 2'b00.	
31:16	Reserved. EAX[31:16] - Return all zeros.	
39:32	ERROR_CODE EDX[7:0] ▪ 0x0 - No Error. ▪ 0x1 - SBFT operation did not start. Other thread could not join. ▪ 0x2 - SBFT operation did not start. Interrupt occurred prior to SBFT coordination. ▪ 0x3 - Reserved. ▪ 0x4 - SBFT operation did not start. Non-valid SBFT BUNDLES in the SBFT_BUNDLE_INDEX. ▪ 0x5 - SBFT operation did not start. Mismatch in arguments between threads T0/T1. ▪ 0x6 - SBFT operation did not start. Core is not capable of performing SBFT currently. ▪ 0x7 - Reserved. ▪ 0x8 - SBFT operation did not start. Exceeded number of Logical Processors (LP) allowed to run SBFT-At-Field concurrently. ▪ 0x9 - SBFT operation did not start. Interrupt occurred or timer about to expire. ▪ 0xA - SBFT operation did not start. SBFT_PGM_INDEX is not valid. ▪ 0xB - SBFT operation aborted due to corrupted chunk. ▪ 0xC - SBFT operation did not start. TAP Data error. ▪ 0xD - SBFT operation did not start. SBFT program is not valid. All other error codes are reserved.	
60:40	Reserved. EDX[28:8] - Return all zeros.	
61	TEST_FAIL EDX[29:29] - Architectural Signature failed. Last thread executed HLT and completed SBFT and EBX != 0xACED.	
63:62	SBFT_STATUS EDX[31:30] - SBFT status (result of running SBAF). ▪ 00 - PASS. ▪ 10 - INTERRUPTED. ▪ 01 - FAILED SIGNATURE CHECK. ▪ 11 - FAILED.	
Register Address: 2BEH, 702	MSR_SBFT_MODULE_ID	
MSR_SBFT_MODULE_ID (R/O)		Module

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:0	SBFT-AT-FIELD_REVID EAX[31:0] - Maps to Revision field in external header (offset 4).	
40:32	CURRENT_STRIDE_INDEX EDX[8:0] - Stride Index.	
63:41	Reserved. EDX[31:9] - Return all zeros.	
Register Address: 2BFH, 703	MSR_SBFTAF_LAST_WP	
MSR_SBFTAF_LAST_WP (R/O)		Module
31:0	LAST_WP EAX[31:0] - Provides information about the core when the last WRMSR(ACTIVATE_SBFT) was executed. Available only if enumerated in INTEGRITY_CAPABILITIES[10:9].	
39:32	Reserved.	
63:40	Reserved. EDX[31:8] - Return all zeros.	
Register Address: 2C2H, 706	MSR_COPY_SCAN_HASHES	
MSR_COPY_SCAN_HASHES (W/O)		Module
63:0	SCAN_HASH-ADDR EDX:EAX contains the linear address of the SCAN Test HASH Binary loaded into memory	
Register Address: 2C3H, 707	MSR_SCAN_HASHES_STATUS	
MSR_SCAN_HASHES_STATUS (R/O)		Core
15:0	CHUNK_SIZE EAX[15:0] - Chunk size of the test in KB.	
31:16	TOTAL_NUM_CHUNKS EAX[31:16] - Total number of chunks.	
39:32	ERROR_CODE EDX[7:0] - The error code refers to the LP that runs WRMSR(2C2H). • 0x0 - Reserved. • 0x1 - Attempt to copy scan-hashes when copy already in progress. • 0x2 - Secure Memory not set up correctly. • 0x3 - Scan-Image Header Image_info.ProgramID does not match MSR_INTEGRITY_CAPABILITIES[31:24], or scan-image header Processor-Signature doesn't match F/M/S, or scan-image header Processor-Flags doesn't match PlatformID. • 0x4 - Reserved. • 0x5 - Integrity check failed. • 0x6 - WRMSR(0x2C6) Re-install of scan test image attempted when current scan test image is in use by other LPs. • 0x7 - Aborted due to #PF (Page Fault). • 0x8 - Unable to generate a Random Value.	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
48:40	NUM_CHUNKS_IN_STRIDE EDX[16:8] - Number of Chunks in stride. This is the number of chunks that are installed. 0 in this field means that the CPU does not support strides, otherwise, the stride value must be >=1.	
50:49	Reserved. EDX[18:17] - Set to all zeros.	
62:51	NAME EDX[30:19] - Maximum Number of cores that can run Intel® In-field Scan simultaneously minus 1. 0 means 1 core at a time.	
63	VALID EDX[31] - Valid bit is set when COPY_SCAN_HASHES completed.	
Register Address: 2C4H, 708	MSR_AUTHENTICATE_AND_COPY_CHUNK	
MSR_AUTHENTICATE_AND_COPY_CHUNK (R/O)		Core
63:0	BASE_CHUNK_TABLE_ADDR EDX:EAX[63:0] - Linear Address pointing to the CHUNK TABLE (TABLE_BASE).	
Register Address: 2C5H, 709	MSR_CHUNKS_AUTHENTICATION_STATUS	
MSR_CHUNKS_AUTHENTICATION_STATUS (R/O)		Core
15:0	VALID_CHUNKS EAX[15:0] - Total number of Valid (authenticated) chunks.	
31:16	NUM_CHUNKS_IN_STRIDE EAX[31:16] - Number of Chunks in Stride.	
39:32	ERROR_CODE EDX[7:0] ▪ 0x0 - No-error reported. ▪ 0x1 - Attempt to authenticate a CHUNK which is already. marked as authentic or is currently being installed by another core. ▪ 0x2 - CHUNK authentication error. HASH of chunk did not match expected value. ▪ 0x3 - Aborted due to #PF (Page Fault). ▪ 0x4 - Chunk Outside the current Stride.	
63:40	Reserved. EDX[31:8] - Set to all zeros.	
Register Address: 2C6H, 710	MSR_ACTIVATE_SCAN	
MSR_ACTIVATE_SCAN (W/O)		Core
15:0	CHUNK_START_INDEX EAX[15:0] - Indicates Chunk Index from which to start.	
31:16	CHUNK_STOP_INDEX EAX[31:16] - Indicates what chunk index to stop at (inclusive).	
62:32	THREAD_WAIT_DELAY EDX[30:0] - TSC based delay to allow threads to rendezvous.	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
63	SIGNAL_MCE EDX[31] ▪ If 1: On scan-error log MC in MC4_STATUS and signal MCE if machine check signaling enabled in MC4_CTL[0]. ▪ If 0: Don't no-logging/no-signaling.	
Register Address: 2C7H, 711	MSR_SCAN_STATUS	
MSR_SCAN_STATUS (R/O)		Core
15:0	CHUNK_NUM EAX[15:0] - SCAN Chunk that was reached.	
31:16	CHUNK_STOP_INDEX EAX[31:16] ▪ Indicates what chunk index to stop at (inclusive). ▪ Maps to same field in WRMSR(ACTIVATE_SCAN).	
39:32	ERROR_CODE EDX[7:0] ▪ 0x0 - No Error. ▪ 0x1 - SCAN operation did not start. Other thread could not join. ▪ 0x2 - SCAN operation did not start. Interrupt occurred prior to SCAN coordination. ▪ 0x3 - SCAN operation did not start. Power Management conditions are inadequate to run SAF. ▪ 0x4 - SCAN operation did not start. Non valid chunks in the range CHUNK_STOP_INDEX : CHUNK_START_INDEX. ▪ 0x5 - SCAN operation did not start. Mismatch in arguments between threads T0/T1. ▪ 0x6 - SCAN operation did not start. Core not capable of performing SCAN currently. ▪ 0x7 - Debug Mode. Scan-At-Field results not to be trusted. ▪ 0x8 - SCAN operation did not start. Exceeded number of Logical Processors (LP) allowed to run Scan-At-Field concurrently. MAX_CORE_LIMIT exceeded. ▪ 0x9 - Interrupt occurred. Scan operation aborted prematurely, not all chunks requested have been executed. ▪ 0xB - Scan operation aborted due to corrupted chunk. ▪ 0xC - Scan operation did not start. All other error codes are reserved.	
61:40	Reserved. EDX[29:8] - Return all zeros.	
62	SCAN_CONTROL_ERROR EDX[30] ▪ SCAN error in the Scan-At-Field controller. ▪ Non ECC error.	
63	SCAN_SIGNATURE_ERROR EDX[31] ▪ SCAN SIGNATURE error in the SCAN pattern fetched from main memory. ▪ Non ECC error.	
Register Address: 2C8H, 712	MSR_SCAN_MODULE_ID	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MSR_SCAN_MODULE_ID (R/O)		Module
31:0	SCAN-AT-FIELD_REVID EAX[31:0] - Maps to Revision field in external header (offset 4).	
40:32	CURRENT_STRIDE_INDEX EDX[8:0] - Stride Index.	
63:41	Reserved. EDX[31:9] - Return all zeros.	
Register Address: 2C9H, 713	MSR_LAST_SAF_WP	
MSR_LAST_SAF_WP (R/O)		Module
31:0	LAST_WP EAX[31:0] ▪ Provides information about the core when the last WRMSR(ACTIVATE_SCAN) was executed. ▪ Available only if enumerated in INTEGRITY_CAPABILITIES[10:9].	
39:32	Reserved. EDX[7:0]	
63:40	Reserved. EDX[31:8] - Return all zeros.	
Register Address: 2D9H, 729	MSR_INTEGRITY_CAPABILITIES	
MSR_INTEGRITY_CAPABILITIES (R/O) Enumerates features supported in Functional Safety.		Thread
0	STARTUP_SCAN_BIST When set to 1, processor supports Startup SCAN BIST.	
1	STARTUP_MEM_BIST When set to 1, processor supports Startup MEM BIST.	
2	PERIODIC_MEM_BIST When set to 1, processor supports Periodic MEM BIST.	
3	LOCKSTEP When set to 1, processor supports Lock Step Mode.	
4	PERIODIC_SCAN_BIST When set to 1, processor supports Periodic SCAN BIST.	
5	PLL_LOSS_DETECT When set to 1, processor supports PLL LOSS detection.	
6	PWR_LOSS_DETECT When set to 1, processor supports Power Loss detection.	
7	PERRINJ When set to 1, processor supports FUSA PERRINJ.	
8	SBFT_AT_FIELD When set to 1, processor supports SBFT-At-Field.	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
10:9	SAF_GEN_REV 00 = REV1; 01 = REV2; 10 = REV3; 11 = REV4.	
14:11	Reserved.	
15	PRESERVE_MEMORY_NEEDED When set to 1, processor supports FUSARR_BASE/MASK MSRs.	
20:16	TID_BIT_SHIFT Number of bits to shift right on x2APICID to get a unique topology ID of all logical processors that share a scan test engine.	
21	ALL_LP_JOIN_NEEDED All logical processors that share scan test engine need to be tested together and must join using MSR_ACTIVATE_SCAN.	
23:22	Reserved.	
31:24	PATTERN_ID Processor scan pattern ID. ID of the startup and periodic scan programs supported for this part.	
63:32	Reserved.	
Register Address: 30CH, 780	IA32_FIXED_CTR3	
Fixed-Function Performance Counter 3 (R/W) See Table 2-2.		Thread
Register Address: 4D0H, 1232	IA32_MCG_EXT_CTL	
IA32_MCG_EXT_CTL (R/W) See Table 2-2.		Thread
Register Address: 4F0H, 1264	MSR_SAF_CTRL	
MSR_SAF_CTRL (W/O)		Core
0	INVALIDATE_CURRENT_STRIDE EAX[0] ▪ Write of 1 invalidates the currently installed stride. ▪ Clears only the VALID_CHUNKS field on a RDMSR(CHUNKS_AUTHENTICATION_STATUS).	
63:1	Reserved.	
Register Address: 4F8H, 1272	MSR_SBFT_CTRL	
MSR_SBFT_CTRL (W/O)		Module
0	INVALIDATE_CURRENT_STRIDE EAX[0] - Write of 1 invalidates the currently installed stride.	
63:1	Reserved. EDX[31:0],EAX[31:1]	
Register Address: 540H, 1344	MSR_THREAD_UARCH_CTL	
Thread Microarchitectural Control (R/W)		Thread

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
0	WB_MEM_STRM_LD_DISABLE Disable streaming behavior for MOVNTDQA loads to WB memory type. If set, these accesses will be treated like regular cacheable loads (Data will be cached).	
63:1	Reserved.	
Register Address: 541H, 1345	MSR_CORE_UARCH_CTL	
Core Microarchitecture Control MSR (R/W)		Core
0	SCRUB_DIS L1 scrubbing disable.	
63:1	Reserved.	
Register Address: 664H, 1636	MSR_MC6_RESIDENCY	
MSR_MC6_RESIDENCY (R/O) Time spent in the Module C6-State. Provided in units compatible to P1 clock frequency (Guaranteed / Maximum Core Non-Turbo Frequency).		Module
63:0	RESIDENCY Time that this module is in module-specific C6 states since last reset.	
Register Address: 6E1H, 1761	IA32_PKRS	
IA32_PKRS (R/W) Specifies the PK permissions associated with each protection domain for supervisor pages. See Table 2-2.		Thread
Register Address: 7A3H, 1955	IA32_MCU_EXT_SERVICE	
MCU Extended Service MSR (R/O) If IA32_ARCH_CAPABILITIES[22] = 1. See Table 2-2.		Module
Register Address: 7A4H, 1956	IA32_MCU_ROLLBACK_MIN_ID	
Minimal MCU Revision ID for Rollback (R/O) See Table 2-2.		Module
Register Address: 7B0H, 1968	IA32_ROLLBACK_SIGN_ID_0	
Rollback ID 0 (R/O) See Table 2-2.		Module
Register Address: 7B1H, 1969	IA32_ROLLBACK_SIGN_ID_1	
Rollback ID 1 (R/O) See Table 2-2.		Module
Register Address: 7B2H, 1970	IA32_ROLLBACK_SIGN_ID_2	
Rollback ID 2 (R/O) See Table 2-2.		Module
Register Address: 7B3H, 1971	IA32_ROLLBACK_SIGN_ID_3	
Rollback ID 3 (R/O) See Table 2-2.		Module
Register Address: 7B4H, 1972	IA32_ROLLBACK_SIGN_ID_4	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Rollback ID 4 (R/O) See Table 2-2.		Module
Register Address: 7B5H, 1973	IA32_ROLLBACK_SIGN_ID_5	
Rollback ID 5 (R/O) See Table 2-2.		Module
Register Address: 7B6H, 1974	IA32_ROLLBACK_SIGN_ID_6	
Rollback ID 6 (R/O) See Table 2-2.		Module
Register Address: 7B7H, 1975	IA32_ROLLBACK_SIGN_ID_7	
Rollback ID 7 (R/O) See Table 2-2.		Module
Register Address: 7B8H, 1976	IA32_ROLLBACK_SIGN_ID_8	
Rollback ID 8 (R/O) See Table 2-2.		Module
Register Address: 7B9H, 1977	IA32_ROLLBACK_SIGN_ID_9	
Rollback ID 9 (R/O) See Table 2-2.		Module
Register Address: 7BAH, 1978	IA32_ROLLBACK_SIGN_ID_10	
Rollback ID 10 (R/O) See Table 2-2.		Module
Register Address: 7BBH, 1979	IA32_ROLLBACK_SIGN_ID_11	
Rollback ID 11 (R/O) See Table 2-2.		Module
Register Address: 7BCH, 1980	IA32_ROLLBACK_SIGN_ID_12	
Rollback ID 12 (R/O) See Table 2-2.		Module
Register Address: 7BDH, 1981	IA32_ROLLBACK_SIGN_ID_13	
Rollback ID 13 (R/O) See Table 2-2.		Module
Register Address: 7BEH, 1982	IA32_ROLLBACK_SIGN_ID_14	
Rollback ID 14 (R/O) See Table 2-2.		Module
Register Address: 7BFH, 1983	IA32_ROLLBACK_SIGN_ID_15	
Rollback ID 15 (R/O) See Table 2-2.		Module
Register Address: 981H, 2433	IA32_TME_CAPABILITY	
IA32_TME_CAPABILITY (R/O) See Table 2-2.		Package
Register Address: 982H, 2434	IA32_TME_ACTIVATE	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32_TME_ACTIVATE (R/W) See Table 2-2.		Package
Register Address: 983H, 2435	IA32_TME_EXCLUDE_MASK	
Intel TME Exclude Mask (R/W) See Table 2-2.		Package
Register Address: 984H, 2436	IA32_TME_EXCLUDE_BASE	
Intel TME Exclude Base (R/W) See Table 2-2.		Package
Register Address: 985H, 2437	IA32_UINTR_RR	
User Interrupt Request Register (R/W) See Table 2-2.		Thread
Register Address: 986H, 2438	IA32_UINTR_HANDLER	
User Interrupt Handler Address (R/W) See Table 2-2.		Thread
Register Address: 987H, 2439	IA32_UINTR_STACKADJUST	
User Interrupt Stack Adjustment (R/W) See Table 2-2.		Thread
Register Address: 988H, 2440	IA32_UINTR_NV	
User-Interrupt Size and Notification Vector (R/W) See Table 2-2.		Thread
Register Address: 989H, 2441	IA32_UINTR_PD	
User Interrupt PID Address (R/W) See Table 2-2.		Thread
Register Address: 98AH, 2442	IA32_UINTR_TT	
User-Interrupt Target Table (R/W) See Table 2-2.		Thread
Register Address: 990H, 2448	IA32_COPY_STATUS	
IA32_COPY_STATUS (R/O) See Table 2-2.		Thread
Register Address: 991H, 2449	IA32_IWKEYBACKUP_STATUS	
IA32_IWKEYBACKUP_STATUS (R/O) See Table 2-2.		Package
Register Address: 9FBH, 2555	IA32_TME_CLEAR_SAVED_KEY	
IA32_TME_CLEAR_SAVED_KEY (R/W) See Table 2-2.		Package
Register Address: 9FFH, 2559	MSR_CORE_MKTME_ACTIVATE	
MSR to read TME_ACTIVATE[MK_TME_KEYID_BITS] (R/O)		Core
31:0	Reserved.	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
35:32	READ_MK_TME_KEYID_BITS This value will be returned on a RDMSR, but must be zero on a WRMSR.	
63:36	Reserved.	
Register Address: C84H, 3204	MSR_MBA_CFG	
Memory Bandwidth Allocation (MBA) Configuration (R/W)		Package
1:0	Reserved.	
2	RAMBAE Resource Aware MBA Enable.	
63:3	Reserved.	
Register Address: CA0H, 3232	MSR_RMID_SNC_CONFIG	
RMID_SNC_CONFIG (R/W)		Package
0	RMID_LOCALIZED_DISTRIBUTION_MODE_ENABLE If set, Localized RMID distribution mode is enabled. If Clear, RMID Sharing mode is enabled.	
63:1	Reserved.	
Register Address: D50H, 3408	IA32_L2_QOS_EXT_BW_THRTL_0	
Memory Bandwidth Enforcement for COS0 (R/W) See Table 2-2.		Package
Register Address: D51H, 3409	IA32_L2_QOS_EXT_BW_THRTL_1	
Memory Bandwidth Enforcement for COS1 (R/W) See Table 2-2.		Package
Register Address: D52H, 3410	IA32_L2_QOS_EXT_BW_THRTL_2	
Memory Bandwidth Enforcement for COS2 (R/W) See Table 2-2.		Package
Register Address: D53H, 3411	IA32_L2_QOS_EXT_BW_THRTL_3	
Memory Bandwidth Enforcement for COS3 (R/W) See Table 2-2.		Package
Register Address: D54H, 3412	IA32_L2_QOS_EXT_BW_THRTL_4	
Memory Bandwidth Enforcement for COS4 (R/W) See Table 2-2.		Package
Register Address: D55H, 3413	IA32_L2_QOS_EXT_BW_THRTL_5	
Memory Bandwidth Enforcement for COS5 (R/W) See Table 2-2.		Package
Register Address: D56H, 3414	IA32_L2_QOS_EXT_BW_THRTL_6	
Memory Bandwidth Enforcement for COS6 (R/W) See Table 2-2.		Package
Register Address: D57H, 3415	IA32_L2_QOS_EXT_BW_THRTL_7	

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Memory Bandwidth Enforcement for COS7 (R/W) See Table 2-2.		Package
Register Address: D58H, 3416	IA32_L2_QOS_EXT_BW_THRTL_8	
Memory Bandwidth Enforcement for COS8 (R/W) See Table 2-2.		Package
Register Address: D59H, 3417	IA32_L2_QOS_EXT_BW_THRTL_9	
Memory Bandwidth Enforcement for COS9 (R/W) See Table 2-2.		Package
Register Address: D5AH, 3418	IA32_L2_QOS_EXT_BW_THRTL_10	
Memory Bandwidth Enforcement for COS10 (R/W) See Table 2-2.		Package
Register Address: D5BH, 3419	IA32_L2_QOS_EXT_BW_THRTL_11	
Memory Bandwidth Enforcement for COS11 (R/W) See Table 2-2.		Package
Register Address: D5CH, 3420	IA32_L2_QOS_EXT_BW_THRTL_12	
Memory Bandwidth Enforcement for COS12 (R/W) See Table 2-2.		Package
Register Address: D5DH, 3421	IA32_L2_QOS_EXT_BW_THRTL_13	
Memory Bandwidth Enforcement for COS13 (R/W) See Table 2-2.		Package
Register Address: D5EH, 3422	IA32_L2_QOS_EXT_BW_THRTL_14	
Memory Bandwidth Enforcement for COS14 (R/W) See Table 2-2.		Package
Register Address: D91H, 3473	IA32_COPY_LOCAL_TO_PLATFORM	
See Table 2-2.		Thread
Register Address: D92H, 3474	IA32_COPY_PLATFORM_TO_LOCAL	
See Table 2-2.		Thread
Register Address: D93H, 3475	IA32_PASID	
See Table 2-2.		Thread
Register Address: 1400H, 5120	IA32_SEAMRR_BASE	
SEAM Memory Range Register for TDx - Base Address (R/W) See Table 2-2.		Core
Register Address: 1401H, 5121	IA32_SEAMRR_MASK	
SEAM Memory Range Register for TDX (R/W) See Table 2-2.		Core
Register Address: 1A8FH, 6799	MSR_STLB_QOS_INFO	
STLB_QOS_INFO (R/O) STLB QoS MASK configuration.		Core

Table 2-56. Additional MSRs Supported by the Intel® Xeon® 6 P-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
5:0	NCLOS Number of CLOS supported for STLB resource using minus-1 notation.	
15:6	Reserved.	
19:16	4K_2M_CBM Length of capacity bitmask for 4K and 2M pages using minus-1 notation.	
28:20	Reserved.	
29	STLB_FILL_TRANSLATION_MSR_SUPPORTED MSR interface to fill STLB translations supported.	
30	4K_2M_ALIAS Indicates that 4K/2M pages alias into the same structure.	
63:31	Reserved.	
Register Address: 1B01H, 6913	IA32_UARCH_MISC_CTL	
IA32_UARCH_MISC_CTL (R/W) See Table 2-2.		Thread

2.17.11 MSRs Introduced in the Intel® Xeon® 6 E-Core Processors

Table 2-57 lists additional MSRs for the Intel Xeon 6 E-core processors. Intel Xeon 6 E-core processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_AFH.

For an MSR listed in Table 2-57 that also appears in the model-specific tables of prior generations, Table 2-57 supersedes prior generation tables.

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 2FH, 47	IA32_BARRIER	
BARRIER (R/O) The IA32_BARRIER MSR ensures ordered execution by acting like LFENCE, controlling the sequencing of subsequent MSR reads after prior MSR reads and instructions. See Table 2-2.		Core
Register Address: 33H, 51	MSR_MEMORY_CONTROL	
Memory Control (R/W) Disables split locks, which are locked instructions that split a cache line.		Core
26:0	Reserved.	
27	UC_STORE_THROTTLE If set to 1, when enabled, the processor allows one in-progress, post-retirement UC stores at a time.	
28	UC_LOCK_DISABLE If set to 1, a UC load lock will trigger a fault. If clear to 0, UC load locks proceed normally.	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
29	SPLIT_LOCK_DISABLE If set to 1, a split lock will trigger an #AC fault. If clear to 0, split locks proceed normally.	
63:30	Reserved.	
Register Address: 34H, 52	MSR_SMI_COUNT	
SMI Counter (R/W)		Thread
31:0	SMI_COUNT Running count of SMI events since the last reset.	
63:32	Reserved.	
Register Address: 39H, 57	MSR_SOCKET_ID	
Socket ID (R/W) Reassigns the package-specific portions of the APIC ID. This MSR is used on scalable DP and high-end MP platforms to resolve legacy-mode APIC ID conflicts.		Package
10:0	PACKAGE_ID Holds package ID. This reflects the upper bits of the APIC ID.	
63:11	Reserved.	
Register Address: 7BH, 123	IA32_MCU_ENUMERATION	
Enumeration of Architectural Features (R/O) See Table 2-2.		Package
Register Address: 7CH, 124	IA32_MCU_STATUS	
MCU Status (R/O) Communicates results from the previous patch loads. See Table 2-2.		Package
Register Address: 87H, 135	IA32_MKTME_KEYID_PARTITIONING	
MKTME KEY ID Partitioning (R/O) Enumerates the number of activated KeyIDs for Intel TME-MK and Intel TDX. See Table 2-2.		Package
Register Address: 98H, 152	MSR_SEAM_WBINVDP	
SEAM WBINVDP (R/W) Allows software to WBINVD sections of the LLC.		Thread
63:0	HANDLE Caches sub-block to invalidate.	
Register Address: 99H, 153	MSR_SEAM_WBNOINVDP	
SEAM WBNOINVDP (R/W) Allows software to WBNOINVD sections of the LLC.		Thread
63:0	HANDLE Caches sub-block to invalidate.	
Register Address: 9AH, 154	MSR_SEAM_INTR_PENDING	
SEAM Interrupt Pending (R/O) Report out some event pending bits.		Thread
0	INTR Interrupt is pending.	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
1	NMI NMI is pending.	
2	SMI SMI is pending.	
4:3	OTHER_EVENTS Other events pending.	
63:5	Reserved.	
Register Address: 9BH, 155	IA32_SMM_MONITOR_CTL	
SMM Monitor Control (R/W) The SMM Monitor Configuration involves SMM code specifying the MSEG location and enabling dual-monitor treatment by writing to the corresponding MSR. See Table 2-2.		Thread
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information (R/O) Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
15:8	MAX_NON_TURBO_LIM_RATIO This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.	
25:16	Reserved.	
26	DCU_16K_MODE_AVAIL 0b: Indicates that the part does not support the 16K DCU mode. 1b: Indicates that the part supports 16K DCU mode.	
27	Reserved.	
28	PRG_TURBO_RATIO_EN Programmable Turbo Ratios per number of Active Cores. 0 = Programming Not Allowed. 1 = Programming Allowed.	
34:29	Reserved.	
35	BIOS_GUARD_ENABLE Indicates whether the BIOS Guard feature is enabled in the CPU.	
36	PEG2DMIDIS_EN 0 = PEG2DMIDIS is disabled. 1 = PEG2DMIDIS is enabled.	
39:37	Reserved.	
47:40	MAX_EFFICIENCY_RATIO Maximum Efficiency Ratio. This is given in units of 100 MHz.	
58:48	Reserved.	
59	SMM_SUPOVR_STATE_LOCK_ENABLE When set, indicates that the CPU supports MSR SMM_SUPOVR_STATE_LOCK and the Hardware Shield feature.	
63:60	Reserved.	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: CFH, 207	IA32_CORE_CAPABILITIES	
IA32 Core Capabilities Register (R/W) If CPUID.(EAX=07H, ECX=0):EDX[30] = 1. This MSR provides an architectural enumeration function for model-specific behavior.		Core
0	STLB_QOS When set to 1, processor supports STLB QoS.	
1	Reserved.	
2	INTEGRITY_SUPPORTED When set to 1, processor supports Functional Safety. Specific FUSA capabilities are enumerated in MSR_FUSA_CAPABILITIES.	
3	RSM_IN_CPL0_ONLY Intel System Resources Defense: When set to 1, RSM will only be allowed in CPL0 and will #GP for all non-CPL0 privilege levels.	
4	UC_LOCK_DISABLE When set to 1, processor supports UC load lock disable.	
5	SPLIT_LOCK_DISABLE When set to 1, processor supports #AC on split locks.	
6	SNP_FILTER_QOS When set to 1, processor supports Snoop Filter Quality of Service MSRs.	
7	UC_STORE_THROTTLING When set to 1, processor supports UC store throttling through MSR_MEMORY_CTRL[UC_STORE_THROTTLE].	
63:8	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: FEH, 254	IA32_MTRRCAP	
Memory Type Range Register (R/O) See Table 2-2.		Core
Register Address: 140H, 320	MSR_FEATURE_ENABLES	
Miscellaneous Enables for Thread-Specific Features (R/W)		Thread
0	AESNI_LOCK Once this bit is set, writes to this register will not be allowed.	
63:1	Reserved.	
Register Address: 1B0H, 432	IA32_ENERGY_PERF_BIAS	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32_ENERGY_PERF_BIAS (R/W) See Table 2-2.		Thread
Register Address: 1B1H, 433	IA32_PACKAGE_THERM_STATUS	
IA32_PACKAGE_THERM_STATUS See Table 2-2.		Package
Register Address: 1B2H, 434	IA32_PACKAGE_THERM_INTERRUPT	
IA32_PACKAGE_THERM_INTERRUPT (R/W) See Table 2-2.		Package
Register Address: 2A1H, 673	MSR_PRMRR_BASE_1	
Processor Reserved Memory Range Register - Physical Base Control Register (R/W)		Core
2:0	MEMTYPE Memory Type for PRMRR accesses.	
3	CONFIGURED PRMRR base configured.	
11:4	Reserved.	
51:12	BASE PRMRR Base address.	
63:52	Reserved.	
Register Address: 2A2H, 674	MSR_PRMRR_BASE_2	
Processor Reserved Memory Range Register - Physical Base Control Register (R/W)		Core
2:0	MEMTYPE Memory Type for PRMRR accesses.	
3	CONFIGURED PRMRR base configured.	
11:4	Reserved.	
51:12	BASE PRMRR Base address.	
63:52	Reserved.	
Register Address: 2A3H, 675	MSR_PRMRR_BASE_3	
Processor Reserved Memory Range Register - Physical Base Control Register (R/W)		Core
2:0	MEMTYPE Memory Type for PRMRR accesses.	
3	CONFIGURED PRMRR base configured.	
11:4	Reserved.	
51:12	BASE PRMRR Base address.	
63:52	Reserved.	
Register Address: 2C2H, 706	MSR_COPY_SCAN_HASHES	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MSR_COPY_SCAN_HASHES (W/O)		Module
63:0	SCAN_HASH-ADDR EDX:EAX contains the linear address of the SCAN Test HASH Binary loaded into memory	
Register Address: 2C3H, 707	MSR_SCAN_HASHES_STATUS	
MSR_SCAN_HASHES_STATUS (R/O)		Core
15:0	CHUNK_SIZE EAX[15:0] - Chunk size of the test in KB.	
31:16	TOTAL_NUM_CHUNKS EAX[31:16] - Total number of chunks.	
39:32	ERROR_CODE EDX[7:0] - The error code refers to the LP that runs WRMSR(2C2H). ▪ 0x0 - Reserved. ▪ 0x1 - Attempt to copy scan-hashes when copy already in progress. ▪ 0x2 - Secure Memory not set up correctly. ▪ 0x3 - Scan-Image Header Image_info.ProgramID does not match MSR_INTEGRITY_CAPABILITIES[31:24], or scan-image header Processor-Signature doesn't match F/M/S, or scan-image header Processor-Flags doesn't match PlatformID. ▪ 0x4 - Reserved. ▪ 0x5 - Integrity check failed. ▪ 0x6 - WRMSR(0x2C6) Re-install of scan test image attempted when current scan test image is in use by other LPs. ▪ 0x7 - Aborted due to #PF (Page Fault). ▪ 0x8 - Unable to generate a Random Value.	
48:40	NUM_CHUNKS_IN_STRIDE EDX[16:8] - Number of Chunks in stride. This is the number of chunks that are installed. 0 in this field means that the CPU does not support strides, otherwise, the stride value must be >=1	
50:49	Reserved. EDX[18:17] - Set to all zeros.	
62:51	NAME EDX[30:19] - Maximum Number of cores that can run Intel® In-field Scan simultaneously minus 1. 0 means 1 core at a time.	
63	VALID EDX[31] - Valid bit is set when COPY_SCAN_HASHES completed.	
Register Address: 2C4H, 708	MSR_AUTHENTICATE_AND_COPY_CHUNK	
MSR_AUTHENTICATE_AND_COPY_CHUNK(R/O)		Core
63:0	BASE_CHUNK_TABLE_ADDR EDX:EAX[63:0] - Linear Address pointing to the CHUNK TABLE (TABLE_BASE).	
Register Address: 2C5H, 709	MSR_CHUNKS_AUTHENTICATION_STATUS	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MSR_CHUNKS_AUTHENTICATION_STATUS (R/O)		Core
15:0	VALID_CHUNKS EAX[15:0] - Total number of Valid (authenticated) chunks.	
31:16	NUM_CHUNKS_IN_STRIDE EAX[31:16] - Number of Chunks in Stride.	
39:32	ERROR_CODE EDX[7:0] ▪ 0x0 - No-error reported. ▪ 0x1 - Attempt to authenticate a CHUNK which is already marked as authentic or is currently being installed by another core. ▪ 0x2 - CHUNK authentication error. HASH of chunk did not match expected value. ▪ 0x3 - Aborted due to #PF (Page Fault). ▪ 0x4 - Chunk Outside the current Stride.	
63:40	Reserved. EDX[31:8] - Set to all zeros.	
Register Address: 2C6H, 710	MSR_ACTIVATE_SCAN	
MSR_ACTIVATE_SCAN (W/O)		Core
15:0	CHUNK_START_INDEX EAX[15:0] - Indicates Chunk Index from which to start.	
31:16	CHUNK_STOP_INDEX EAX[31:16] - Indicates what chunk index to stop at (inclusive).	
62:32	THREAD_WAIT_DELAY EDX[30:0] - TSC based delay to allow threads to rendezvous.	
63	SIGNAL_MCE EDX[31] ▪ If 1: On scan-error log MC in MC4_STATUS and signal MCE if machine check signaling enabled in MC4_CTL[0]. ▪ If 0: Don't no-logging/no-signaling.	
Register Address: 2C7H, 711	MSR_SCAN_STATUS	
MSR_SCAN_STATUS (R/O)		Core
15:0	CHUNK_NUM EAX[15:0] - SCAN Chunk that was reached.	
31:16	CHUNK_STOP_INDEX EAX[31:16] ▪ Indicates what chunk index to stop at (inclusive). ▪ Maps to same field in WRMSR(ACTIVATE_SCAN).	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
39:32	ERROR_CODE EDX[7:0] ▪ 0x0 - No Error. ▪ 0x1 - SCAN operation did not start. Other thread could not join. ▪ 0x2 - SCAN operation did not start. Interrupt occurred prior to SCAN coordination. ▪ 0x3 - SCAN operation did not start. Power Management conditions are inadequate to run SAF. ▪ 0x4 - SCAN operation did not start. Non valid chunks in the range CHUNK_STOP_INDEX : CHUNK_START_INDEX. ▪ 0x5 - SCAN operation did not start. Mismatch in arguments between threads T0/T1. ▪ 0x6 - SCAN operation did not start. Core not capable of performing SCAN currently. ▪ 0x7 - Debug Mode. Scan-At-Field results not to be trusted. ▪ 0x8 - SCAN operation did not start. Exceeded number of Logical Processors (LP) allowed to run Scan-At-Field concurrently. MAX_CORE_LIMIT exceeded. ▪ 0x9 - Interrupt occurred. Scan operation aborted prematurely, not all chunks requested have been executed. ▪ 0xB - Scan operation aborted due to corrupted chunk. ▪ 0xC - Scan operation did not start. All other error codes are reserved.	
61:40	Reserved. EDX[29:8] - Return all zeros.	
62	SCAN_CONTROL_ERROR EDX[30] ▪ SCAN error in the Scan-At-Field controller. ▪ Non ECC error.	
63	SCAN_SIGNATURE_ERROR EDX[31] ▪ SCAN SIGNATURE error in the SCAN pattern fetched from main memory. ▪ Non ECC error.	
Register Address: 2C8H, 712	MSR_SCAN_MODULE_ID	
MSR_SCAN_MODULE_ID (R/O)		Module
31:0	SCAN-AT-FIELD_REVID EAX[31:0] - Maps to Revision field in external header (offset 4).	
40:32	CURRENT_STRIDE_INDEX EDX[8:0] - Stride Index.	
63:41	Reserved. EDX[31:9] - Return all zeros.	
Register Address: 2C9H, 713	MSR_LAST_SAF_WP	
MSR_LAST_SAF_WP (R/O)		Module

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:0	LAST_WP EAX[31:0] ▪ Provides information about the core when the last WRMSR(ACTIVATE_SCAN) was executed. ▪ Available only if enumerated in INTEGRITY_CAPABILITIES[10:9].	
39:32	Reserved. EDX[7:0]	
63:40	Reserved. EDX[31:8] - Return all zeros.	
Register Address: 2D6H, 726	MSR_TRIGGER_PERIODIC_MEM_BIST	
MSR_TRIGGER_PERIODIC_MEM_BIST (W/O)		Core
0	SIGNAL_MCE EAX[0] - If 1, then signal MCE on fail if machine check signaling enabled in the corresponding MCi_CTL. If 0 then don't signal machine checks.	
7:1	ARRAY_BANK EAX[7:1] - Reserved.	
15:8	TST_STEP_PARAM EAX[15:8] 0: Test All Arrays, or Test Arrays in STEPs of NUM_STEPS.	
31:16	Reserved. EAX[31:16]	
63:32	Reserved. EAX[31:0]	
Register Address: 2D7H, 727	MSR_PERIODIC_MEM_BIST_STATUS	
MSR_PERIODIC_MEM_BIST_STATUS (R/O)		Core
0	MEM_BIST_STATUS 0: PASS. 1: FAIL.	
63:1	Reserved.	
Register Address: 2D9H, 729	MSR_INTEGRITY_CAPABILITIES	
MSR_INTEGRITY_CAPABILITIES (R/O) Enumerates features supported in Functional Safety.		Thread
0	STARTUP_SCAN_BIST When set to 1, processor supports Startup SCAN BIST.	
1	STARTUP_MEM_BIST When set to 1, processor supports Startup MEM BIST.	
2	PERIODIC_MEM_BIST When set to 1, processor supports Periodic MEM BIST.	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
3	LOCKSTEP When set to 1, processor supports Lock Step Mode.	
4	PERIODIC_SCAN_BIST When set to 1, processor supports Periodic SCAN BIST.	
5	PLL_LOSS_DETECT When set to 1, processor supports PLL LOSS detection.	
6	PWR_LOSS_DETECT When set to 1, processor supports Power Loss detection.	
7	PERRINJ When set to 1, processor supports FUSA PERRINJ.	
8	SBFT_AT_FIELD When set to 1, processor supports SBFT-At-Field.	
10:9	SAF_GEN_REV 00 = REV1; 01 = REV2; 10 = REV3; 11 = REV4.	
14:11	Reserved.	
15	PRESERVE_MEMORY_NEEDED When set to 1, processor supports FUSARR_BASE/MASK MSRs.	
20:16	TID_BIT_SHIFT Number of bits to shift right on x2APICID to get a unique topology ID of all logical processors that share a scan test engine.	
21	ALL_LP_JOIN_NEEDED All logical processors that share scan test engine need to be tested together and must join using MSR_ACTIVATE_SCAN.	
23:22	Reserved.	
31:24	PATTERN_ID Processor scan pattern ID. ID of the startup and periodic scan programs supported for this part.	
63:32	Reserved.	
Register Address: 2DCH, 732	IA32_INTEGRITY_STATUS	
IA32_INTEGRITY_STATUS (R/O) Provides status information for integrity features. See Table 2-2.		Thread
Register Address: 3F9H, 1017	MSR_PKG_C6_RESIDENCY	
MSR_PKG_C6_RESIDENCY (R/O)		Package
63:0	Package C6 Residency Counter	
Register Address: 3FAH, 1018	MSR_PKG_C7_RESIDENCY	
MSR_PKG_C7_RESIDENCY (R/O)		Package
63:0	Package C7 Residency Counter	
Register Address: 3FCH, 1020	MSR_CORE_C3_RESIDENCY	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MSR_CORE_C3_RESIDENCY (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C3 Residency Counter Time spent in the Core C-State. Provided in units compatible to P1 clock frequency (Guaranteed / Maximum Core Non-Turbo Frequency).	
Register Address: 3FDH, 1021	MSR_CORE_C6_RESIDENCY	
MSR_CORE_C6_RESIDENCY (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C6 Residency Counter Time spent in the Core C-State. Provided in units compatible to P1 clock frequency (Guaranteed / Maximum Core Non-Turbo Frequency).	
Register Address: 3FEH, 1022	MSR_CORE_C7_RESIDENCY	
MSR_CORE_C7_RESIDENCY (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Core
63:0	CORE C7 Residency Counter Time spent in the Core C-State. Provided in units compatible to P1 clock frequency (Guaranteed / Maximum Core Non-Turbo Frequency).	
Register Address: 4F0H, 1264	MSR_SAF_CTRL	
MSR_SAF_CTRL (W/O)		Core
0	INVALIDATE_CURRENT_STRIDE EAX[0] ▪ Write of 1 invalidates the currently installed stride. ▪ Clears only the VALID_CHUNKS field on a RDMSR(CHUNKS_AUTHENTICATION_STATUS).	
63:1	Reserved.	
Register Address: 664H, 1636	MSR_MC6_RESIDENCY	
MSR_MC6_RESIDENCY (R/O) Time spent in the Module C6-State. Provided in units compatible to P1 clock frequency (Guaranteed / Maximum Core Non-Turbo Frequency).		Module
63:0	RESIDENCY Time that this module is in module-specific C6 states since last reset.	
Register Address: 6E0H, 1760	IA32_TSC_DEADLINE	
TSC Target of Local APIC's TSC Deadline Mode (R/W) See Table 2-2.		Thread
Register Address: 7A3H, 1955	IA32_MCU_EXT_SERVICE	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MCU Extended Service (R/W) See Table 2-2.		Module
Register Address: 7A4H, 1956	IA32_MCU_ROLLBACK_MIN_ID	
Minimal MCU Revision ID (R/O) See Table 2-2.		Module
Register Address: 7A5H, 1957	IA32_MCU_STAGING_MBOX_ADDR	
IA32_MCU_STAGING_MBOX_ADDR (R/O) See Table 2-2.		Package
Register Address: 7B0H, 1968	IA32_ROLLBACK_SIGN_ID_0	
Rollback ID 0 (R/O) See Table 2-2.		Module
Register Address: 7B1H, 1969	IA32_ROLLBACK_SIGN_ID_1	
Rollback ID 1 (R/O) See Table 2-2.		Module
Register Address: 7B2H, 1970	IA32_ROLLBACK_SIGN_ID_2	
Rollback ID 2 (R/O) See Table 2-2.		Module
Register Address: 7B3H, 1971	IA32_ROLLBACK_SIGN_ID_3	
Rollback ID 3 (R/O) See Table 2-2.		Module
Register Address: 7B4H, 1972	IA32_ROLLBACK_SIGN_ID_4	
Rollback ID 4 (R/O) See Table 2-2.		Module
Register Address: 7B5H, 1973	IA32_ROLLBACK_SIGN_ID_5	
Rollback ID 5 (R/O) See Table 2-2.		Module
Register Address: 7B6H, 1974	IA32_ROLLBACK_SIGN_ID_6	
Rollback ID 6 (R/O) See Table 2-2.		Module
Register Address: 7B7H, 1975	IA32_ROLLBACK_SIGN_ID_7	
Rollback ID 7 (R/O) See Table 2-2.		Module
Register Address: 7B8H, 1976	IA32_ROLLBACK_SIGN_ID_8	
Rollback ID 8 (R/O) See Table 2-2.		Module
Register Address: 7B9H, 1977	IA32_ROLLBACK_SIGN_ID_9	
Rollback ID 9 (R/O) See Table 2-2.		Module
Register Address: 7BAH, 1978	IA32_ROLLBACK_SIGN_ID_10	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Rollback ID 10 (R/O) See Table 2-2.		Module
Register Address: 7BBH, 1979	IA32_ROLLBACK_SIGN_ID_11	
Rollback ID 11 (R/O) See Table 2-2.		Module
Register Address: 7BCH, 1980	IA32_ROLLBACK_SIGN_ID_12	
Rollback ID 12 (R/O) See Table 2-2.		Module
Register Address: 7BDH, 1981	IA32_ROLLBACK_SIGN_ID_13	
Rollback ID 13 (R/O) See Table 2-2.		Module
Register Address: 7BEH, 1982	IA32_ROLLBACK_SIGN_ID_14	
Rollback ID 14 (R/O) See Table 2-2.		Module
Register Address: 7BFH, 1983	IA32_ROLLBACK_SIGN_ID_15	
Rollback ID 15 (R/O) See Table 2-2.		Module
Register Address: 988H, 2440	IA32_UINTR_NV	
User Interrupt Size and Notification Vector (R/W) See Table 2-2.		Thread
Register Address: 9FBH, 2555	IA32_TME_CLEAR_SAVED_KEY	
IA32_TME_CLEAR_SAVED_KEY (R/W) See Table 2-2.		Package
Register Address: 9FFH, 2559	MSR_CORE_MKTME_ACTIVATE	
MSR_CORE_MKTME_ACTIVATE (R/O) MSR to read TME_ACTIVATE[MK_TME_KEYID_BITS].		Core
31:0	Reserved.	
35:32	READ_MK_TME_KEYID_BITS This value will be returned on a RDMSR, but must be zero on a WRMSR.	
63:36	Reserved.	
Register Address: C84H, 3204	MSR_MBA_CFG	
Memory Bandwidth Allocation (MBA) Configuration (R/W)		Package
1:0	Reserved.	
2	RAMBAE Resource Aware MBA Enable.	
63:3	Reserved.	
Register Address: CA0H, 3232	MSR_RMID_SNC_CONFIG	
MSR_RMID_SNC_CONFIG (R/W)		Package

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
0	RMID_LOCALIZED_DISTRIBUTION_MODE_ENABLE If set, Localized RMID distribution mode is enabled. If Clear, RMID Sharing mode is enabled.	
63:1	Reserved.	
Register Address: D50H, 3408	IA32_L2_QOS_EXT_BW_THRTL_0	
Memory Bandwidth Enforcement for COS0 (R/W) See Table 2-2.		Package
Register Address: D51H, 3409	IA32_L2_QOS_EXT_BW_THRTL_1	
Memory Bandwidth Enforcement for COS1 (R/W) See Table 2-2.		Package
Register Address: D52H, 3410	IA32_L2_QOS_EXT_BW_THRTL_2	
Memory Bandwidth Enforcement for COS2 (R/W) See Table 2-2.		Package
Register Address: D53H, 3411	IA32_L2_QOS_EXT_BW_THRTL_3	
Memory Bandwidth Enforcement for COS3 (R/W) See Table 2-2.		Package
Register Address: D54H, 3412	IA32_L2_QOS_EXT_BW_THRTL_4	
Memory Bandwidth Enforcement for COS4 (R/W) See Table 2-2.		Package
Register Address: D55H, 3413	IA32_L2_QOS_EXT_BW_THRTL_5	
Memory Bandwidth Enforcement for COS5 (R/W) See Table 2-2.		Package
Register Address: D56H, 3414	IA32_L2_QOS_EXT_BW_THRTL_6	
Memory Bandwidth Enforcement for COS6 (R/W) See Table 2-2.		Package
Register Address: D57H, 3415	IA32_L2_QOS_EXT_BW_THRTL_7	
Memory Bandwidth Enforcement for COS7 (R/W) See Table 2-2.		Package
Register Address: D58H, 3416	IA32_L2_QOS_EXT_BW_THRTL_8	
Memory Bandwidth Enforcement for COS8 (R/W) See Table 2-2.		Package
Register Address: D59H, 3417	IA32_L2_QOS_EXT_BW_THRTL_9	
Memory Bandwidth Enforcement for COS9 (R/W) See Table 2-2.		Package
Register Address: D5AH, 3418	IA32_L2_QOS_EXT_BW_THRTL_10	
Memory Bandwidth Enforcement for COS10 (R/W) See Table 2-2.		Package
Register Address: D5BH, 3419	IA32_L2_QOS_EXT_BW_THRTL_11	

Table 2-57. Additional MSRs Supported by the Intel® Xeon® 6 E-Core Processors (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Memory Bandwidth Enforcement for COS11 (R/W) See Table 2-2.		Package
Register Address: D5CH, 3420	IA32_L2_QOS_EXT_BW_THRTL_12	
Memory Bandwidth Enforcement for COS12 (R/W) See Table 2-2.		Package
Register Address: D5DH, 3421	IA32_L2_QOS_EXT_BW_THRTL_13	
Memory Bandwidth Enforcement for COS13 (R/W) See Table 2-2.		Package
Register Address: D5EH, 3422	IA32_L2_QOS_EXT_BW_THRTL_14	
Memory Bandwidth Enforcement for COS14 (R/W) See Table 2-2.		Package
Register Address: E00H, 3584	IA32_QOS_CORE_BW_THRTL_0	
CBA Levels Based on COS for Bandwidth Throttling (R/W) See Table 2-2.		Thread
Register Address: E01H, 3585	IA32_QOS_CORE_BW_THRTL_1	
CBA Levels Based on COS for Bandwidth Throttling (R/W) See Table 2-2.		Thread
Register Address: 1400H, 5120	IA32_SEAMRR_BASE	
SEAM Memory Range Register for TDX - Base Address (R/W) See Table 2-2.		Core
Register Address: 1401H, 5121	IA32_SEAMRR_MASK	
SEAM Memory Range Register for TDX (R/W) See Table 2-2.		Core
Register Address: 1A8FH, 6799	MSR_STLB_QOS_INFO	
STLB_QOS_INFO (R/O) STLB QoS MASK configuration.		Core
5:0	NCLOS Number of CLOS supported for STLB resource using minus-1 notation.	
15:6	Reserved.	
19:16	4K_2M_CBM Length of capacity bitmask for 4K and 2M pages using minus-1 notation.	
28:20	Reserved.	
29	STLB_FILL_TRANSLATION_MSR_SUPPORTED MSR interface to fill STLB translations supported.	
30	4K_2M_ALIAS Indicates that 4K/2M pages alias into the same structure.	
63:31	Reserved.	

2.17.12 MSRs Introduced in the Intel® Series 2 Core™ Ultra Processor Supporting Performance Hybrid Architecture

Table 2-58 lists additional MSRs for the Intel Series 2 Core Ultra processor with a CPUID Signature DisplayFamily_DisplayModel value of 06_BDH. Table 2-59 lists the MSRs unique to the processor P-core. Table 2-60 lists the MSRs unique to the processor E-core.

For an MSR listed in Table 2-58, Table 2-59, or Table 2-60 that also appears in the model-specific tables of prior generations, Table 2-58, Table 2-59, and Table 2-60 supersede prior generation tables.

Table 2-58. Additional MSRs Supported by the Intel® Series 2 Core™ Ultra Processors Supporting Performance Hybrid Architecture

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Thread
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation (R/W) See Table 2-2.		Thread
Register Address: 601H, 1537	MSR_PKG_POWER_LIMIT_4	
Package Power Limit 4 (R/W) Package-level maximum power limit (in Watts).		Package
15:0	POWER_LIMIT_4 PL4 Value in 0.125 W increments. This field is locked by PKG_POWER_LIMIT_4.LOCK. When the LOCK bit is set to 1, this field becomes Read Only. If the value is 0, PL4 limit is disabled.	
30:16	Reserved.	
31	LOCK This bit will lock the POWER_LIMIT_4 settings in this register and will also lock this setting. This means that once set to 1, the POWER_LIMIT_4 setting and this bit become Read Only until the next Warm Reset.	
63:32	Reserved.	
Register Address: 630H, 1584	MSR_PKG_C8_RESIDENCY	
MSR_PKG_C8_RESIDENCY (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
59:0	Package C8 Residency Counter Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.	
63:60	Reserved.	
Register Address: 631H, 1585	MSR_PKG_C9_RESIDENCY	
MSR_PKG_C9_RESIDENCY (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package

Table 2-58. Additional MSRs Supported by the Intel® Series 2 Core™ Ultra Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
59:0	Package C9 Residency Counter Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.	
63:60	Reserved.	
Register Address: 632H, 1586	MSR_PKG_C10_RESIDENCY	
MSR_PKG_C10_RESIDENCY (R/O) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.		Package
59:0	Package C10 Residency Counter Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.	
63:60	Reserved.	
Register Address: 651H, 1617	MSR_SECONDARY_TURBO_RATIO_LIMIT_CORES	
SECONDARY_TURBO_RATIO_LIMIT_CORES (R/W) This register defines the active core ranges for each frequency point. ▪ NUMCORE[0:7] must be populated in ascending order. ▪ NUMCORE[i+1] must be greater than NUMCORE[i]. ▪ Entries with NUMCORE[i] == 0 will be ignored. ▪ The last valid entry must have NUMCORE >= the number of cores in the SKU. If any of the rules above are broken, we will silently reject the configuration.		Package
7:0	CORE_COUNT_0 Defines the active core ranges for each frequency point.	
15:8	CORE_COUNT_1 Defines the active core ranges for each frequency point.	
23:16	CORE_COUNT_2 Defines the active core ranges for each frequency point.	
31:24	CORE_COUNT_3 Defines the active core ranges for each frequency point.	
39:32	CORE_COUNT_4 Defines the active core ranges for each frequency point.	
47:40	CORE_COUNT_5 Defines the active core ranges for each frequency point.	
55:48	CORE_COUNT_6 Defines the active core ranges for each frequency point.	
63:56	CORE_COUNT_7 Defines the active core ranges for each frequency point.	
Register Address: 658H, 1624	MSR_WEIGHTED_CORE_C0	
Core-Count Weighted C0 Residency (R/O)		Package

Table 2-58. Additional MSRs Supported by the Intel® Series 2 Core™ Ultra Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:0	DATA Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.	
Register Address: 659H, 1625	MSR_ANY_CORE_C0	
Any Core C0 Residency (R/O)		Package
63:0	DATA Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.	
Register Address: 65AH, 1626	MSR_ANY_GFXE_C0	
Any Graphics Engine C0 Residency (R/O)		Package
63:0	DATA Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.	
Register Address: 65BH, 1627	MSR_CORE_GFXE_OVERLAP_C0	
Core and Graphics Engine Overlapped C0 Residency (R/O)		Package
63:0	DATA Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.	
Register Address: C88H, 3208	IA32_RESOURCE_PRIORITY	
Thread scope Resource Priority Enable (R/W) See Table 2-2.		Thread
Register Address: C89H, 3209	IA32_RESOURCE_PRIORITY_PKG	
IA32_RESOURCE_PRIORITY_PKG (R/W) See Table 2-2.		Package
Register Address: 1900H, 6400	IA32_PMC_GP0_CTR	
Full Width Writable General Performance Counter 0 (R/W) See Table 2-2.		Thread
Register Address: 1901H, 6401	IA32_PMC_GP0_CFG_A	
IA32_PMC_GP0_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 0. See Table 2-2.		Thread
Register Address: 1904H, 6404	IA32_PMC_GP1_CTR	
Full Width Writable General Performance Counter 1 (R/W) See Table 2-2.		Thread
Register Address: 1905H, 6405	IA32_PMC_GP1_CFG_A	

Table 2-58. Additional MSRs Supported by the Intel® Series 2 Core™ Ultra Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32_PMC_GP1_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 1. See Table 2-2.		Thread
Register Address: 1908H, 6408	IA32_PMC_GP2_CTR	
Full Width Writable General Performance Counter 2 (R/W) See Table 2-2.		Thread
Register Address: 1909H, 6409	IA32_PMC_GP2_CFG_A	
IA32_PMC_GP2_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 2. See Table 2-2.		Thread
Register Address: 190CH, 6412	IA32_PMC_GP3_CTR	
Full Width Writable General Performance Counter 3 (R/W) See Table 2-2.		Thread
Register Address: 190DH, 6413	IA32_PMC_GP3_CFG_A	
IA32_PMC_GP3_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 3. See Table 2-2.		Thread
Register Address: 1910H, 6416	IA32_PMC_GP4_CTR	
Full Width Writable General Performance Counter 4 (R/W) See Table 2-2.		Thread
Register Address: 1911H, 6417	IA32_PMC_GP4_CFG_A	
IA32_PMC_GP4_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 4. See Table 2-2.		Thread
Register Address: 1914H, 6420	IA32_PMC_GP5_CTR	
Full Width Writable General Performance Counter 5 (R/W) See Table 2-2.		Thread
Register Address: 1915H, 6421	IA32_PMC_GP5_CFG_A	
IA32_PMC_GP5_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 5. See Table 2-2.		Thread
Register Address: 1918H, 6424	IA32_PMC_GP6_CTR	
Full Width Writable General Performance Counter 6 (R/W) See Table 2-2.		Thread
Register Address: 1919H, 6425	IA32_PMC_GP6_CFG_A	
IA32_PMC_GP6_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 6. See Table 2-2.		Thread
Register Address: 191CH, 6428	IA32_PMC_GP7_CTR	

Table 2-58. Additional MSRs Supported by the Intel® Series 2 Core™ Ultra Processors Supporting Performance Hybrid Architecture (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Full Width Writable General Performance Counter 7 (R/W) See Table 2-2.		Thread
Register Address: 191DH, 6429	IA32_PMC_GP7_CFG_A	
IA32_PMC_GP7_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 7. See Table 2-2.		Thread
Register Address: 1980H, 6528	IA32_PMC_FX0_CTR	
IA32_PMC_FX0_CTR (R/W) Fixed-Function Performance Counter 0 - Instructions Retired. See Table 2-2.		Thread
Register Address: 1984H, 6532	IA32_PMC_FX1_CTR	
IA32_PMC_FX1_CTR (R/W) Fixed-Function Performance Counter 1 - Unhalted core clock cycles. See Table 2-2.		Thread
Register Address: 1988H, 6536	IA32_PMC_FX2_CTR	
IA32_PMC_FX2_CTR (R/W) Fixed-Function Performance Counter 2 - Unhalted core reference cycles. See Table 2-2.		Thread

The MSRs listed in Table 2-59 are unique to the Intel Series 2 Core Ultra processor P-core. These MSRs are not supported on the processor E-core.

Table 2-59. MSRs Supported by the Intel® Series 2 Core™ Ultra Processor P-core

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: C9H, 201	IA32_PMC8	
General Performance Counter 8 (R/W) See Table 2-2.		Thread
Register Address: CAH, 202	IA32_PMC9	
General Performance Counter 9 (R/W) See Table 2-2.		Thread
Register Address: 18EH, 398	IA32_PERFEVTSEL8	
Performance Event Select Register 8 (R/W) See Table 2-2.		Thread
Register Address: 18FH, 399	IA32_PERFEVTSEL9	
Performance Event Select Register 9 (R/W) See Table 2-2.		Thread
Register Address: 30CH, 780	IA32_FIXED_CTR3	
Fixed-Function Performance Counter 3 (R/W) See Table 2-2.		Thread
Register Address: 329H, 809	MSR_PERF_METRICS	

Table 2-59. MSRs Supported by the Intel® Series 2 Core™ Ultra Processor P-core (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
MSR_PERF_METRICS (R/W) This register provides built-in support for Top-down Micro-architecture Analysis (TMA) metrics. It exposes the four TMA Level 1 metrics where the lower 32 bits are divided into four 8 bit fields, each of which is an integer percentage of the total TOPDOWN.SLOTS (as reported by fixed-function counter 3).		Thread
7:0	RETIRING Percent of utilized by uops that eventually retire (commit).	
15:8	BAD_SPECULATION Percent of Wasted due to incorrect speculation, covering Utilized by uops that do not retire, or Recovery Bubbles (unutilized slots).	
23:16	FRONTEND_BOUND Percent of Unutilized slots where Front-end did not deliver a uop while Back-end is ready.	
31:24	BACKEND_BOUND Percent of Unutilized slots where a uop was not delivered to Back-end due to lack of Back-end resources.	
39:32	MULTI_UOPS Frontend bound.	
47:40	BRANCH_MISPREDICTS Frontend bound.	
55:48	FRONTEND_LATENCY Frontend bound.	
63:56	MEMORY_BOUND Frontend bound.	
Register Address: 4C9H, 1225	IA32_A_PMC8	
Full Width Writable IA32_PMC8 Alias (R/W) See Table 2-2.		Thread
Register Address: 4CAH, 1226	IA32_A_PMC9	
Full Width Writable IA32_PMC9 Alias (R/W) See Table 2-2.		Thread
Register Address: 540H, 1344	MSR_THREAD_UARCH_CTL	
Thread Uarch Control (R/W)		Thread
0	WB_MEM_STRM_LD_DISABLE Disable streaming behavior for MOVNTDQA loads to WB memory type. If set, these accesses will be treated like regular cacheable loads (Data will be cached).	
63:1	Reserved.	
Register Address: 540H, 1344	MSR_CORE_UARCH_CTL	
Core Uarch Control (R/W)		Core
0	SCRUB_DIS L1 scrubbing disable.	
63:1	Reserved.	

Table 2-59. MSRs Supported by the Intel® Series 2 Core™ Ultra Processor P-core (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1920H, 6432	IA32_PMC_GP8_CTR	
Full Width Writable General Performance Counter 8 (R/W) See Table 2-2.		Thread
Register Address: 1921H, 6433	IA32_PMC_GP8_CFG_A	
IA32_PMC_GP8_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 8. See Table 2-2.		Thread
Register Address: 1924H, 6436	IA32_PMC_GP9_CTR	
Full Width Writable General Performance Counter 9 (R/W) See Table 2-2.		Thread
Register Address: 1925H, 6437	IA32_PMC_GP9_CFG_A	
IA32_PMC_GP9_CFG_A (R/W) Performance Event Select Register used to control the operation of the General Performance Counter 9. See Table 2-2.		Thread
Register Address: 198CH, 6540	IA32_PMC_FX3_CTR	
IA32_PMC_FX3_CTR (R/W) See Table 2-2.		Thread

The MSRs listed in Table 2-60 are unique to the Intel Series 2 Core Ultra processor E-core. These MSRs are not supported on the processor P-core.

Table 2-60. MSRs Supported by the Intel® Series 2 Core™ Ultra Processor E-core

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 2DCH, 732	IA32_INTEGRITY_STATUS	
Status Information for Integrity Features (R/O) See Table 2-2.		Thread
Register Address: 30DH, 781	IA32_FIXED_CTR4	
Fixed-Function Performance Counter 4 - Top-down Bad Speculation (R/W) See Table 2-2.		Thread
Register Address: 30EH, 782	IA32_FIXED_CTR5	
Fixed-Function Performance Counter 5 - Top-down Frontend Bound (R/W) See Table 2-2.		Thread
Register Address: 30FH, 783	IA32_FIXED_CTR6	
Fixed-Function Performance Counter 6 - Top-down Retiring (R/W) See Table 2-2.		Thread
Register Address: D18H, 3352	IA32_L2_MASK_8	
L2 CAT Mask for COS8 (R/W) See Table 2-2.		Module
Register Address: D19H, 3353	IA32_L2_MASK_9	

Table 2-60. MSRs Supported by the Intel® Series 2 Core™ Ultra Processor E-core (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
L2 CAT Mask for COS9 (R/W) See Table 2-2.		Module
Register Address: D1AH, 3354	IA32_L2_MASK_10	
L2 CAT Mask for COS10 (R/W) See Table 2-2.		Module
Register Address: D1BH, 3355	IA32_L2_MASK_11	
L2 CAT Mask for COS11 (R/W) See Table 2-2.		Module
Register Address: D1CH, 3356	IA32_L2_MASK_12	
L2 CAT Mask for COS12 (R/W) See Table 2-2.		Module
Register Address: D1DH, 3357	IA32_L2_MASK_13	
L2 CAT Mask for COS13 (R/W) See Table 2-2.		Module
Register Address: D1EH, 3358	IA32_L2_MASK_14	
L2 CAT Mask for COS14 (R/W) See Table 2-2.		Module
Register Address: D1FH, 3359	IA32_L2_MASK_15	
L2 CAT Mask for COS15 (R/W) See Table 2-2.		Module
Register Address: 1878H, 6264	MSR_WORK_CONSERVING_CLOS	
Work Conserving CLOS (R/W)		Module
0	WC_VALID WC Valid Bit that indicates WC MSR has been setup. This bit must be set for the WC algorithm to be enabled.	
7:1	Reserved.	
11:8	CLOS_START_PRI1 Starting CLOS range for priority 1.	
15:12	CLOS_END_PRI1 Ending CLOS range for priority 1.	
19:16	CLOS_START_PRI2 Starting CLOS range for priority 2.	
23:20	CLOS_END_PRI2 Ending CLOS range for priority 2.	
27:24	CLOS_START_PRI3 Starting CLOS range for priority 3.	
31:28	CLOS_END_PRI3 Ending CLOS range for priority 3.	
63:32	Reserved.	
Register Address: 1903H, 6403	IA32_PMC_GP0_CFG_C	

Table 2-60. MSRs Supported by the Intel® Series 2 Core™ Ultra Processor E-core (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32_PMC_GP0_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 1907H, 6407	IA32_PMC_GP1_CFG_C	
IA32_PMC_GP1_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 190AH, 6410	IA32_PMC_GP2_CFG_B	
IA32_PMC_GP2_CFG_B (R/W) See Table 2-2.		Thread
Register Address: 190BH, 6411	IA32_PMC_GP2_CFG_C	
IA32_PMC_GP2_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 190EH, 6414	IA32_PMC_GP3_CFG_B	
IA32_PMC_GP3_CFG_B (R/W) See Table 2-2.		Thread
Register Address: 190FH, 6415	IA32_PMC_GP3_CFG_C	
IA32_PMC_GP3_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 1912H, 6418	IA32_PMC_GP4_CFG_B	
IA32_PMC_GP4_CFG_B (R/W) See Table 2-2.		Thread
Register Address: 1913H, 6419	IA32_PMC_GP4_CFG_C	
IA32_PMC_GP4_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 1916H, 6422	IA32_PMC_GP5_CFG_B	
IA32_PMC_GP5_CFG_B (R/W) See Table 2-2.		Thread
Register Address: 1917H, 6423	IA32_PMC_GP5_CFG_C	
IA32_PMC_GP5_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 191AH, 6426	IA32_PMC_GP6_CFG_B	
IA32_PMC_GP6_CFG_B (R/W) See Table 2-2.		Thread
Register Address: 191BH, 6427	IA32_PMC_GP6_CFG_C	
IA32_PMC_GP6_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 191EH, 6430	IA32_PMC_GP7_CFG_B	
IA32_PMC_GP7_CFG_B (R/W) See Table 2-2.		Thread
Register Address: 191FH, 6431	IA32_PMC_GP7_CFG_C	

Table 2-60. MSRs Supported by the Intel® Series 2 Core™ Ultra Processor E-core (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
IA32_PMC_GP7_CFG_C (R/W) See Table 2-2.		Thread
Register Address: 1982H, 6530	IA32_PMC_FX0_CFG_B	
Fixed-Function Counter Reload Configuration Register (R/W) See Table 2-2.		Thread
Register Address: 1983H, 6531	IA32_PMC_FX0_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 0 (R/W) See Table 2-2.		Thread
Register Address: 1986H, 6534	IA32_PMC_FX1_CFG_B	
Fixed-Function Counter Reload Configuration Register (R/W) See Table 2-2.		Thread
Register Address: 1987H, 6535	IA32_PMC_FX1_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 1 (R/W) See Table 2-2.		Thread
Register Address: 198BH, 6539	IA32_PMC_FX2_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 2 (R/W) See Table 2-2.		Thread
Register Address: 1990H, 6544	IA32_PMC_FX4_CTR	
Fixed-Function Performance Counter 4 - Top-down Bad Speculation (R/W) See Table 2-2.		Thread
Register Address: 1993H, 6547	IA32_PMC_FX4_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 4 (R/W) See Table 2-2.		Thread
Register Address: 1994H, 6548	IA32_PMC_FX5_CTR	
Fixed-Function Performance Counter 5 - Top-down Frontend Bound (R/W) See Table 2-2.		Thread
Register Address: 1997H, 6551	IA32_PMC_FX5_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 5 (R/W) See Table 2-2.		Thread
Register Address: 1998H, 6552	IA32_PMC_FX6_CTR	
Fixed-Function Performance Counter 5 - Top-down Bad Retiring (R/W) See Table 2-2.		Thread
Register Address: 199BH, 6555	IA32_PMC_FX6_CFG_C	
Extended Perf Event Selector for Fixed-Function Counter 6 (R/W) See Table 2-2.		Thread

2.18 MSRS IN THE INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES AND THE INTEL® XEON PHI™ PROCESSOR 7215/7285/7295 SERIES

The Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with a CPUID Signature DisplayFamily_DisplayModel value of 06_57H, supports the MSR interfaces listed in Table 2-61. These processors are based on the Knights Landing microarchitecture. The Intel® Xeon Phi™ processor 7215, 7285, 7295 series, with a CPUID Signature DisplayFamily_DisplayModel value of 06_85H, supports the MSR interfaces listed in Table 2-61 and Table 2-62. These processors are based on the Knights Mill microarchitecture. Some MSRs are shared between a pair of processor cores, and the scope is marked as module.

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 0H, 0	IA32_P5_MC_ADDR	
See Section 2.23, “MSRs in Pentium Processors.”		Module
Register Address: 1H, 1	IA32_P5_MC_TYPE	
See Section 2.23, “MSRs in Pentium Processors.”		Module
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, “Monitor/Mwait Address Range Determination.” See Table 2-2.		Thread
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER	
See Section 19.17, “Time-Stamp Counter,” and Table 2-2.		Thread
Register Address: 17H, 23	IA32_PLATFORM_ID	
Platform ID (R) See Table 2-2.		Package
Register Address: 1BH, 27	IA32_APIC_BASE	
See Section 12.4.4, “Local APIC Status and Location,” and Table 2-2.		Thread
Register Address: 34H, 52	MSR_SMI_COUNT	
SMI Counter (R/O)		Thread
31:0	SMI Count (R/O)	
63:32	Reserved.	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in Intel 64Processor (R/W) See Table 2-2.		Thread
0	Lock. (R/WL)	
1	Reserved.	
2	Enable VMX outside SMX operation. (R/WL)	
Register Address: 3BH, 59	IA32_TSC_ADJUST	
Per-Logical-Processor TSC ADJUST (R/W) See Table 2-2.		Thread
Register Address: 4EH, 78	IA32_PPIN_CTL (MSR_PPIN_CTL)	
Protected Processor Inventory Number Enable Control (R/W)		Package
0	LockOut (R/WO) See Table 2-2.	

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
1	Enable_PPIN (R/W) See Table 2-2.	
63:2	Reserved	
Register Address: 4FH, 79	IA32_PPIN (MSR_PPIN)	
Protected Processor Inventory Number (R/O)		Package
63:0	Protected Processor Inventory Number (R/O) See Table 2-2.	
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG	
BIOS Update Trigger Register (W) See Table 2-2.		Core
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Thread
Register Address: C1H, 193	IA32_PMC0	
Performance Counter Register See Table 2-2.		Thread
Register Address: C2H, 194	IA32_PMC1	
Performance Counter Register See Table 2-2.		Thread
Register Address: CEH, 206	MSR_PLATFORM_INFO	
Platform Information Contains power management and other model specific features enumeration. See http://biosbits.org.		Package
7:0	Reserved.	
15:8	Maximum Non-Turbo Ratio (R/O) This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.	Package
27:16	Reserved.	
28	Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.	Package
29	Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.	Package
39:30	Reserved.	
47:40	Maximum Efficiency Ratio (R/O) This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.	Package
63:48	Reserved.	

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: E2H, 226	MSR_PKG_CST_CONFIG_CONTROL	
C-State Configuration Control (R/W)		Package
2:0	Package C-State Limit (R/W) Specifies the lowest C-state for the package. This feature does not limit the processor core C-state. The power-on default value from bit[2:0] of this register reports the deepest package C-state the processor is capable to support when manufactured. It is recommended that BIOS always read the power-on default value reported from this bit field to determine the supported deepest C-state on the processor and leave it as default without changing it. 000b - C0/C1 (No package C-state support) 001b - C2 010b - C6 (non retention)* 011b - C6 (Retention)* 100b - Reserved 101b - Reserved 110b - Reserved 111b - No package C-state limit. All C-States supported by the processor are available. Note: C6 retention mode provides more power saving than C6 non-retention mode. Limiting the package to C6 non retention mode does prevent the MSR_PKG_C6_RESIDENCY counter (MSR 3F9h) from being incremented.	
9:3	Reserved.	
10	I/O MWAIT Redirection Enable (R/W) When set, will map IO_read instructions sent to IO registers at MSR_PMG_IO_CAPTURE_BASE[15:0] to MWAIT instructions.	
14:11	Reserved.	
15	CFG Lock (R/O) When set, locks bits [15:0] of this register for further writes until the next reset occurs.	
25	Reserved.	
26	C1 State Auto Demotion Enable (R/W) When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.	
27	Reserved.	
28	C1 State Auto Undemotion Enable (R/W) When set, enables Undemotion from Demoted C1.	
29	PKG C-State Auto Demotion Enable (R/W) When set, enables Package C state demotion.	
63:30	Reserved.	
Register Address: E4H, 228	MSR_PMG_IO_CAPTURE_BASE	
Power Management IO Capture Base (R/W)		Tile

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
15:0	LVL_2 Base Address (R/W) Microcode will compare IO-read zone to this base address to determine if an MWAIT(C2/3/4) needs to be issued instead of the IO-read. Should be programmed to the chipset Plevel_2 IO address.	
22:16	C-State Range (R/W) The IO-port block size in which IO-redirection will be executed (0-127). Should be programmed based on the number of LVLx registers existing in the chipset.	
63:23	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Thread
Register Address: FEH, 254	IA32_MTRRCAP	
Memory Type Range Register (R/O) See Table 2-2.		Core
Register Address: 13CH, 316	MSR_FEATURE_CONFIG	
AES Configuration (RW-L) Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.		Core
1:0	AES Configuration (RW-L) Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, the AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.	
63:2	Reserved.	
Register Address: 140H, 320	MISC_FEATURE_ENABLES	
MISC_FEATURE_ENABLES		Thread
0	Reserved.	
1	User Mode MONITOR and MWAIT (R/W) If set to 1, the MONITOR and MWAIT instructions do not cause invalid-opcode exceptions when executed with CPL > 0 or in virtual-8086 mode. If MWAIT is executed when CPL > 0 or in virtual-8086 mode, and if EAX indicates a C-state other than C0 or C1, the instruction operates as if EAX indicated the C-state C1.	
63:2	Reserved.	
Register Address: 174H, 372	IA32_SYSENTER_CS	
See Table 2-2.		Thread

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 175H, 373	IA32_SYSENTER_ESP	
See Table 2-2.		Thread
Register Address: 176H, 374	IA32_SYSENTER_EIP	
See Table 2-2.		Thread
Register Address: 179H, 377	IA32_MCG_CAP	
See Table 2-2.		Thread
Register Address: 17AH, 378	IA32_MCG_STATUS	
See Table 2-2.		Thread
Register Address: 17DH, 381	MSR_SMM_MCA_CAP	
Enhanced SMM Capabilities (SMM-RO) Reports SMM capability Enhancement. Accessible only while in SMM.		Thread
31:0	Bank Support (SMM-RO) One bit per MCA bank. If the bit is set, that bank supports Enhanced MCA (Default all 0; does not support EMCA).	
55:32	Reserved.	
56	Targeted SMI (SMM-RO) Set if targeted SMI is supported.	
57	SMM_CPU_SVRSTR (SMM-RO) Set if SMM SRAM save/restore feature is supported.	
58	SMM_CODE_ACCESS_CHK (SMM-RO) Set if SMM code access check feature is supported.	
59	Long_Flow_Indication (SMM-RO) If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.	
63:60	Reserved.	
Register Address: 186H, 390	IA32_PERFEVTSEL0	
Performance Monitoring Event Select Register (R/W) See Table 2-2.		Thread
7:0	Event Select.	
15:8	UMask.	
16	USR.	
17	OS.	
18	Edge.	
19	PC.	
20	INT.	
21	AnyThread.	
22	EN.	
23	INV.	

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
31:24	CMASK.	
63:32	Reserved.	
Register Address: 187H, 391	IA32_PERFEVTSEL1	
See Table 2-2.		Thread
Register Address: 198H, 408	IA32_PERF_STATUS	
See Table 2-2.		Package
Register Address: 199H, 409	IA32_PERF_CTL	
See Table 2-2.		Thread
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation (R/W) See Table 2-2.		Thread
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) See Table 2-2.		Module
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2.		Module
0	Thermal Status (R/O)	
1	Thermal Status Log (R/WC0)	
2	PROTCHOT # or FORCEPR# Status (R/O)	
3	PROTCHOT # or FORCEPR# Log (R/WC0)	
4	Critical Temperature Status (R/O)	
5	Critical Temperature Status Log (R/WC0)	
6	Thermal Threshold #1 Status (R/O)	
7	Thermal Threshold #1 Log (R/WC0)	
8	Thermal Threshold #2 Status (R/O)	
9	Thermal Threshold #2 Log (R/WC0)	
10	Power Limitation Status (R/O)	
11	Power Limitation Log (RWC0)	
15:12	Reserved.	
22:16	Digital Readout (R/O)	
26:23	Reserved.	
30:27	Resolution in Degrees Celsius (R/O)	
31	Reading Valid (R/O)	
63:32	Reserved.	
Register Address: 1A0H, 416	IA32_MISC_ENABLE	

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Enable Misc. Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		Thread
0	Fast-Strings Enable	
2:1	Reserved.	
3	Automatic Thermal Control Circuit Enable (R/W)	
6:4	Reserved.	
7	Performance Monitoring Available (R)	
10:8	Reserved.	
11	Branch Trace Storage Unavailable (R/O)	
12	Processor Event Based Sampling Unavailable (R/O)	
15:13	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W)	
18	ENABLE MONITOR FSM (R/W)	
21:19	Reserved.	
22	Limit CPUID Maxval (R/W)	
23	xTPR Message Disable (R/W)	
33:24	Reserved.	
34	XD Bit Disable (R/W) See Table 2-3.	
37:35	Reserved.	
38	Turbo Mode Disable (R/W)	
63:39	Reserved.	
Register Address: 1A2H, 418	MSR_TEMPERATURE_TARGET	
Temperature Target		Package
15:0	Reserved.	
23:16	Temperature Target (R)	
29:24	Target Offset (R/W)	
63:30	Reserved.	
Register Address: 1A4H, 420	MSR_MISC_FEATURE_CONTROL	
Miscellaneous Feature Control (R/W)		
0	DCU Hardware Prefetcher Disable (R/W) If 1, disables the L1 data cache prefetcher.	Core
1	L2 Hardware Prefetcher Disable (R/W) If 1, disables the L2 hardware prefetcher.	Core
63:2	Reserved.	
Register Address: 1A6H, 422	MSR_OFFCORE_RSP_0	
Offcore Response Event Select Register (R/W)		Shared

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 1A7H, 423	MSR_OFFCORE_RSP_1	
Offcore Response Event Select Register (R/W)		Shared
Register Address: 1ADH, 429	MSR_TURBO_RATIO_LIMIT	
Maximum Ratio Limit of Turbo Mode for Groups of Cores (R/W)		Package
0	Reserved.	
7:1	Maximum Number of Cores in Group 0 Number active processor cores which operates under the maximum ratio limit for group 0.	Package
15:8	Maximum Ratio Limit for Group 0 Maximum turbo ratio limit when the number of active cores are not more than the group 0 maximum core count.	Package
20:16	Number of Incremental Cores Added to Group 1 Group 1, which includes the specified number of additional cores plus the cores in group 0, operates under the group 1 turbo max ratio limit = “group 0 Max ratio limit” - “group ratio delta for group 1”.	Package
23:21	Group Ratio Delta for Group 1 An unsigned integer specifying the ratio decrement relative to the Max ratio limit to Group 0.	Package
28:24	Number of Incremental Cores Added to Group 2 Group 2, which includes the specified number of additional cores plus all the cores in group 1, operates under the group 2 turbo max ratio limit = “group 1 Max ratio limit” - “group ratio delta for group 2”.	Package
31:29	Group Ratio Delta for Group 2 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 1.	Package
36:32	Number of Incremental Cores Added to Group 3 Group 3, which includes the specified number of additional cores plus all the cores in group 2, operates under the group 3 turbo max ratio limit = “group 2 Max ratio limit” - “group ratio delta for group 3”.	Package
39:37	Group Ratio Delta for Group 3 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 2.	Package
44:40	Number of Incremental Cores Added to Group 4 Group 4, which includes the specified number of additional cores plus all the cores in group 3, operates under the group 4 turbo max ratio limit = “group 3 Max ratio limit” - “group ratio delta for group 4”.	Package
47:45	Group Ratio Delta for Group 4 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 3.	Package
52:48	Number of Incremental Cores Added to Group 5 Group 5, which includes the specified number of additional cores plus all the cores in group 4, operates under the group 5 turbo max ratio limit = “group 4 Max ratio limit” - “group ratio delta for group 5”.	Package

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
55:53	Group Ratio Delta for Group 5 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 4.	Package
60:56	Number of Incremental Cores Added to Group 6 Group 6, which includes the specified number of additional cores plus all the cores in group 5, operates under the group 6 turbo max ratio limit = “group 5 Max ratio limit” - “group ratio delta for group 6”.	Package
63:61	Group Ratio Delta for Group 6 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 5.	Package
Register Address: 1B0H, 432	IA32_ENERGY_PERF_BIAS	
See Table 2-2.		Thread
Register Address: 1B1H, 433	IA32_PACKAGE_THERM_STATUS	
See Table 2-2.		Package
Register Address: 1B2H, 434	IA32_PACKAGE_THERM_INTERRUPT	
See Table 2-2.		Package
Register Address: 1C8H, 456	MSR_LBR_SELECT	
Last Branch Record Filtering Select Register (R/W) See Section 19.9.2, “Filtering of Last Branch Records.”		Thread
0	CPL_EQ_0	
1	CPL_NEQ_0	
2	JCC	
3	NEAR_REL_CALL	
4	NEAR_IND_CALL	
5	NEAR_RET	
6	NEAR_IND_JMP	
7	NEAR_REL_JMP	
8	FAR_BRANCH	
63:9	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.		Thread
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W)		Thread
0	LBR Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
1	BTF Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	
5:2	Reserved.	
6	TR Setting this bit to 1 enables branch trace messages to be sent.	
7	BTS Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	
8	BTINT When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	
9	BTS_OFF_OS When set, BTS or BTM is skipped if CPL = 0.	
10	BTS_OFF_USR When set, BTS or BTM is skipped if CPL > 0.	
11	FREEZE_LBRS_ON_PMI When set, the LBR stack is frozen on a PMI request.	
12	FREEZE_PERFMON_ON_PMI When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request.	
13	Reserved.	
14	FREEZE_WHILE_SMM When set, freezes PerfMon and trace messages while in SMM.	
31:15	Reserved.	
Register Address: 1DDH, 477	MSR_LER_FROM_LIP	
Last Exception Record from Linear IP (R)		Thread
Register Address: 1DEH, 478	MSR_LER_TO_LIP	
Last Exception Record to Linear IP (R)		Thread
Register Address: 1F2H, 498	IA32_SMRR_PHYSBASE	
See Table 2-2.		Core
Register Address: 1F3H, 499	IA32_SMRR_PHYSMASK	
See Table 2-2.		Core
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0	
See Table 2-2.		Core
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0	
See Table 2-2.		Core
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1	

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
See Table 2-2.		Core
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1	
See Table 2-2.		Core
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2	
See Table 2-2.		Core
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2	
See Table 2-2.		Core
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3	
See Table 2-2.		Core
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3	
See Table 2-2.		Core
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4	
See Table 2-2.		Core
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4	
See Table 2-2.		Core
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5	
See Table 2-2.		Core
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5	
See Table 2-2.		Core
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6	
See Table 2-2.		Core
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6	
See Table 2-2.		Core
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7	
See Table 2-2.		Core
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7	
See Table 2-2.		Core
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000	
See Table 2-2.		Core
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000	
See Table 2-2.		Core
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000	
See Table 2-2.		Core
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000	
See Table 2-2.		Core
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000	
See Table 2-2.		Core

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000	
See Table 2-2.		Core
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000	
See Table 2-2.		Core
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000	
See Table 2-2.		Core
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000	
See Table 2-2.		Core
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000	
See Table 2-2.		Core
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000	
See Table 2-2.		Core
Register Address: 277H, 631	IA32_PAT	
See Table 2-2.		Core
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE	
Default Memory Types (R/W) See Table 2-2.		Core
Register Address: 309H, 777	IA32_FIXED_CTR0	
Fixed-Function Performance Counter Register 0 (R/W) See Table 2-2.		Thread
Register Address: 30AH, 778	IA32_FIXED_CTR1	
Fixed-Function Performance Counter Register 1 (R/W) See Table 2-2.		Thread
Register Address: 30BH, 779	IA32_FIXED_CTR2	
Fixed-Function Performance Counter Register 2 (R/W) See Table 2-2.		Thread
Register Address: 345H, 837	IA32_PERF_CAPABILITIES	
See Table 2-2. See Section 19.4.1, "IA32_DEBUGCTL MSR."		Package
Register Address: 38DH, 909	IA32_FIXED_CTR_CTRL	
Fixed-Function-Counter Control Register (R/W) See Table 2-2.		Thread
Register Address: 38EH, 910	IA32_PERF_GLOBAL_STATUS	
See Table 2-2.		Thread
Register Address: 38FH, 911	IA32_PERF_GLOBAL_CTRL	
See Table 2-2.		Thread
Register Address: 390H, 912	IA32_PERF_GLOBAL_OVF_CTRL	
See Table 2-2.		Thread

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)	
See Table 2-2.		Thread
Register Address: 3F8H, 1016	MSR_PKG_C3_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Package
63:0	Package C3 Residency Counter (R/O)	
Register Address: 3F9H, 1017	MSR_PKG_C6_RESIDENCY	
63:0	Package C6 Residency Counter (R/O)	Package
Register Address: 3FAH, 1018	MSR_PKG_C7_RESIDENCY	
63:0	Package C7 Residency Counter (R/O)	Package
Register Address: 3FCH, 1020	MSR_MC0_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Module
63:0	Module C0 Residency Counter (R/O)	
Register Address: 3FDH, 1021	MSR_MC6_RESIDENCY	
63:0	Module C6 Residency Counter (R/O)	Module
Register Address: 3FFH, 1023	MSR_CORE_C6_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Core
63:0	CORE C6 Residency Counter (R/O)	
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 408H, 1032	IA32_MC2_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 40CH, 1036	IA32_MC3_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 40DH, 1037	IA32_MC3_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 40EH, 1038	IA32_MC3_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Core
Register Address: 410H, 1040	IA32_MC4_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Core
Register Address: 411H, 1041	IA32_MC4_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Core
Register Address: 412H, 1042	IA32_MC4_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Core
Register Address: 414H, 1044	IA32_MC5_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Package
Register Address: 415H, 1045	IA32_MC5_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Package
Register Address: 416H, 1046	IA32_MC5_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs."		Package
Register Address: 4C1H, 1217	IA32_A_PMC0	
See Table 2-2.		Thread
Register Address: 4C2H, 1218	IA32_A_PMC1	
See Table 2-2.		Thread
Register Address: 600H, 1536	IA32_DS_AREA	
DS Save Area (R/W) See Table 2-2.		Thread
Register Address: 606H, 1542	MSR_RAPL_POWER_UNIT	
Unit Multipliers Used in RAPL Interfaces (R/O)		Package
3:0	Power Units See Section 16.10.1, "RAPL Interfaces."	Package
7:4	Reserved.	Package
12:8	Energy Status Units Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).	Package
15:13	Reserved.	Package
19:16	Time Units See Section 16.10.1, "RAPL Interfaces."	Package

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
63:20	Reserved.	
Register Address: 60DH, 1549	MSR_PKG_C2_RESIDENCY	
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.		Package
63:0	Package C2 Residency Counter (R/O)	
Register Address: 610H, 1552	MSR_PKG_POWER_LIMIT	
PKG RAPL Power Limit Control (R/W) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 611H, 1553	MSR_PKG_ENERGY_STATUS	
PKG Energy Status (R/O) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 613H, 1555	MSR_PKG_PERF_STATUS	
PKG Perf Status (R/O) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 614H, 1556	MSR_PKG_POWER_INFO	
PKG RAPL Parameters (R/W) See Section 16.10.3, “Package RAPL Domain.”		Package
Register Address: 618H, 1560	MSR_DRAM_POWER_LIMIT	
DRAM RAPL Power Limit Control (R/W) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 619H, 1561	MSR_DRAM_ENERGY_STATUS	
DRAM Energy Status (R/O) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 61BH, 1563	MSR_DRAM_PERF_STATUS	
DRAM Performance Throttling Status (R/O) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 61CH, 1564	MSR_DRAM_POWER_INFO	
DRAM RAPL Parameters (R/W) See Section 16.10.5, “DRAM RAPL Domain.”		Package
Register Address: 638H, 1592	MSR_PP0_POWER_LIMIT	
PP0 RAPL Power Limit Control (R/W) See Section 16.10.4, “PP0/PP1 RAPL Domains.”		Package
Register Address: 639H, 1593	MSR_PP0_ENERGY_STATUS	
PP0 Energy Status (R/O) See Section 16.10.4, “PP0/PP1 RAPL Domains.”		Package
Register Address: 648H, 1608	MSR_CONFIG_TDP_NOMINAL	
Base TDP Ratio (R/O) See Table 2-25.		Package

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 649H, 1609	MSR_CONFIG_TDP_LEVEL1	
ConfigTDP Level 1 ratio and power level (R/O) See Table 2-25.		Package
Register Address: 64AH, 1610	MSR_CONFIG_TDP_LEVEL2	
ConfigTDP Level 2 ratio and power level (R/O) See Table 2-25.		Package
Register Address: 64BH, 1611	MSR_CONFIG_TDP_CONTROL	
ConfigTDP Control (R/W) See Table 2-25.		Package
Register Address: 64CH, 1612	MSR_TURBO_ACTIVATION_RATIO	
ConfigTDP Control (R/W) See Table 2-25.		Package
Register Address: 690H, 1680	MSR_CORE_PERF_LIMIT_REASONS	
Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)		Package
0	PROCHOT Status (R0)	
1	Thermal Status (R0)	
5:2	Reserved.	
6	VR Therm Alert Status (R0)	
7	Reserved.	
8	Electrical Design Point Status (R0)	
63:9	Reserved.	
Register Address: 6E0H, 1760	IA32_TSC_DEADLINE	
TSC Target of Local APIC's TSC Deadline Mode (R/W) See Table 2-2.		Core
Register Address: 802H, 2050	IA32_X2APIC_APICID	
x2APIC ID Register (R/O)		Thread
Register Address: 803H, 2051	IA32_X2APIC_VERSION	
x2APIC Version Register (R/O)		Thread
Register Address: 808H, 2056	IA32_X2APIC_TPR	
x2APIC Task Priority Register (R/W)		Thread
Register Address: 80AH, 2058	IA32_X2APIC_PPR	
x2APIC Processor Priority Register (R/O)		Thread
Register Address: 80BH, 2059	IA32_X2APIC_EOI	
x2APIC EOI Register (W/O)		Thread
Register Address: 80DH, 2061	IA32_X2APIC_LDR	
x2APIC Logical Destination Register (R/O)		Thread

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 80FH, 2063	IA32_X2APIC_SIVR	
x2APIC Spurious Interrupt Vector Register (R/W)		Thread
Register Address: 810H, 2064	IA32_X2APIC_ISR0	
x2APIC In-Service Register Bits [31:0] (R/O)		Thread
Register Address: 811H, 2065	IA32_X2APIC_ISR1	
x2APIC In-Service Register Bits [63:32] (R/O)		Thread
Register Address: 812H, 2066	IA32_X2APIC_ISR2	
x2APIC In-Service Register Bits [95:64] (R/O)		Thread
Register Address: 813H, 2067	IA32_X2APIC_ISR3	
x2APIC In-Service Register Bits [127:96] (R/O)		Thread
Register Address: 814H, 2068	IA32_X2APIC_ISR4	
x2APIC In-Service Register Bits [159:128] (R/O)		Thread
Register Address: 815H, 2069	IA32_X2APIC_ISR5	
x2APIC In-Service Register Bits [191:160] (R/O)		Thread
Register Address: 816H, 2070	IA32_X2APIC_ISR6	
x2APIC In-Service Register Bits [223:192] (R/O)		Thread
Register Address: 817H, 2071	IA32_X2APIC_ISR7	
x2APIC In-Service Register Bits [255:224] (R/O)		Thread
Register Address: 818H, 2072	IA32_X2APIC_TMR0	
x2APIC Trigger Mode Register Bits [31:0] (R/O)		Thread
Register Address: 819H, 2073	IA32_X2APIC_TMR1	
x2APIC Trigger Mode Register Bits [63:32] (R/O)		Thread
Register Address: 81AH, 2074	IA32_X2APIC_TMR2	
x2APIC Trigger Mode Register Bits [95:64] (R/O)		Thread
Register Address: 81BH, 2075	IA32_X2APIC_TMR3	
x2APIC Trigger Mode Register Bits [127:96] (R/O)		Thread
Register Address: 81CH, 2076	IA32_X2APIC_TMR4	
x2APIC Trigger Mode Register Bits [159:128] (R/O)		Thread
Register Address: 81DH, 2077	IA32_X2APIC_TMR5	
x2APIC Trigger Mode Register Bits [191:160] (R/O)		Thread
Register Address: 81EH, 2078	IA32_X2APIC_TMR6	
x2APIC Trigger Mode Register Bits [223:192] (R/O)		Thread
Register Address: 81FH, 2079	IA32_X2APIC_TMR7	
x2APIC Trigger Mode Register Bits [255:224] (R/O)		Thread
Register Address: 820H, 2080	IA32_X2APIC_IRR0	
x2APIC Interrupt Request Register Bits [31:0] (R/O)		Thread
Register Address: 821H, 2081	IA32_X2APIC_IRR1	

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
x2APIC Interrupt Request Register Bits [63:32] (R/O)		Thread
Register Address: 822H, 2082	IA32_X2APIC_IRR2	
x2APIC Interrupt Request Register Bits [95:64] (R/O)		Thread
Register Address: 823H, 2083	IA32_X2APIC_IRR3	
x2APIC Interrupt Request Register Bits [127:96] (R/O)		Thread
Register Address: 824H, 2084	IA32_X2APIC_IRR4	
x2APIC Interrupt Request Register Bits [159:128] (R/O)		Thread
Register Address: 825H, 2085	IA32_X2APIC_IRR5	
x2APIC Interrupt Request Register Bits [191:160] (R/O)		Thread
Register Address: 826H, 2086	IA32_X2APIC_IRR6	
x2APIC Interrupt Request Register Bits [223:192] (R/O)		Thread
Register Address: 827H, 2087	IA32_X2APIC_IRR7	
x2APIC Interrupt Request Register Bits [255:224] (R/O)		Thread
Register Address: 828H, 2088	IA32_X2APIC_ESR	
x2APIC Error Status Register (R/W)		Thread
Register Address: 82FH, 2095	IA32_X2APIC_LVT_CMCI	
x2APIC LVT Corrected Machine Check Interrupt Register (R/W)		Thread
Register Address: 830H, 2096	IA32_X2APIC_ICR	
x2APIC Interrupt Command Register (R/W)		Thread
Register Address: 832H, 2098	IA32_X2APIC_LVT_TIMER	
x2APIC LVT Timer Interrupt Register (R/W)		Thread
Register Address: 833H, 2099	IA32_X2APIC_LVT_THERMAL	
x2APIC LVT Thermal Sensor Interrupt Register (R/W)		Thread
Register Address: 834H, 2100	IA32_X2APIC_LVT_PMI	
x2APIC LVT Performance Monitor Register (R/W)		Thread
Register Address: 835H, 2101	IA32_X2APIC_LVT_LINT0	
x2APIC LVT LINT0 Register (R/W)		Thread
Register Address: 836H, 2102	IA32_X2APIC_LVT_LINT1	
x2APIC LVT LINT1 Register (R/W)		Thread
Register Address: 837H, 2103	IA32_X2APIC_LVT_ERROR	
x2APIC LVT Error Register (R/W)		Thread
Register Address: 838H, 2104	IA32_X2APIC_INIT_COUNT	
x2APIC Initial Count Register (R/W)		Thread
Register Address: 839H, 2105	IA32_X2APIC_CUR_COUNT	
x2APIC Current Count Register (R/O)		Thread
Register Address: 83EH, 2110	IA32_X2APIC_DIV_CONF	
x2APIC Divide Configuration Register (R/W)		Thread

Table 2-61. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 83FH, 2111	IA32_X2APIC_SELF_IPI	
x2APIC Self IPI Register (W/O)		Thread
Register Address: C000_0080H	IA32_EFER	
Extended Feature Enables See Table 2-2.		Thread
Register Address: C000_0081H	IA32_STAR	
System Call Target Address (R/W) See Table 2-2.		Thread
Register Address: C000_0082H	IA32_LSTAR	
IA-32e Mode System Call Target Address (R/W) See Table 2-2.		Thread
Register Address: C000_0084H	IA32_FMASK	
System Call Flag Mask (R/W) See Table 2-2.		Thread
Register Address: C000_0100H	IA32_FS_BASE	
Map of BASE Address of FS (R/W) See Table 2-2.		Thread
Register Address: C000_0101H	IA32_GS_BASE	
Map of BASE Address of GS (R/W) See Table 2-2.		Thread
Register Address: C000_0102H	IA32_KERNEL_GS_BASE	
Swap Target of BASE Address of GS (R/W) See Table 2-2.		Thread
Register Address: C000_0103H	IA32_TSC_AUX	
AUXILIARY TSC Signature (R/W) See Table 2-2		Thread

Table 2-62 lists model-specific registers that are supported by the Intel® Xeon Phi™ processor 7215, 7285, 7295 series based on the Knights Mill microarchitecture.

Table 2-62. Additional MSRs Supported by the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_85H

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Register Address: 9BH, 155	IA32_SMM_MONITOR_CTL	
SMM Monitor Configuration (R/W) This MSR is readable only if VMX is enabled, and writeable only if VMX is enabled and in SMM mode, and is used to configure the VMX MSEG base address. See Table 2-2.		Core
Register Address: 480H, 1152	IA32_VMX_BASIC	

Table 2-62. Additional MSRs Supported by the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2.		Core
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	
Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2.		Core
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)		Core
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	
Capability Reporting Register of VM-exit Controls (R/O) See Table 2-2.		Core
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-entry Controls (R/O) See Table 2-2.		Core
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2.		Core
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2.		Core
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2.		Core
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2.		Core
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2.		Core
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	
Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2.		Core
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2	
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Table 2-2.		Core
Register Address: 48CH, 1164	IA32_VMX_EPT_VPID_ENUM	
Capability Reporting Register of EPT and VPID (R/O) See Table 2-2.		Core
Register Address: 48DH, 1165	IA32_VMX_TRUE_PINBASED_CTLS	

Table 2-62. Additional MSRs Supported by the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_85H (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Scope**
Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O) See Table 2-2.		Core
Register Address: 48EH, 1166	IA32_VMX_TRUE_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O) See Table 2-2.		Core
Register Address: 48FH, 1167	IA32_VMX_TRUE_EXIT_CTLS	
Capability Reporting Register of VM-Exit Flex Controls (R/O) See Table 2-2.		Core
Register Address: 490H, 1168	IA32_VMX_TRUE_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Flex Controls (R/O) See Table 2-2.		Core
Register Address: 491H, 1169	IA32_VMX_FMFUNC	
Capability Reporting Register of VM-Function Controls (R/O) See Table 2-2.		Core

2.19 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 2-63 lists MSRs (architectural and model-specific) that are defined across processor generations based on Intel NetBurst microarchitecture. The processor can be identified by its CPUID signatures of DisplayFamily encoding of 0FH, see Table 2-1.

- MSRs with an "IA32_" prefix are designated as "architectural." This means that the functions of these MSRs and their addresses remain the same for succeeding families of IA-32 processors.
- MSRs with an "MSR_" prefix are model specific with respect to address functionalities. The column "Model Availability" lists the model encoding value(s) within the Pentium 4 and Intel Xeon processor family at the specified register address. The model encoding value of a processor can be queried using CPUID. See "CPUID—CPU Identification" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique**[1]
Register Address: 0H, 0	IA32_P5_MC_ADDR		
See Section 2.23, "MSRs in Pentium Processors."		0, 1, 2, 3, 4, 6	Shared
Register Address: 1H, 1	IA32_P5_MC_TYPE		
See Section 2.23, "MSRs in Pentium Processors."		0, 1, 2, 3, 4, 6	Shared
Register Address: 6H, 6	IA32_MONITOR_FILTER_LINE_SIZE		
See Section 10.10.5, "Monitor/Mwait Address Range Determination."		3, 4, 6	Shared
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER		
Time Stamp Counter See Table 2-2.		0, 1, 2, 3, 4, 6	Unique

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
On earlier processors, only the lower 32 bits are writable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writable.			
Register Address: 17H, 23	IA32_PLATFORM_ID		
Platform ID (R) See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.		0, 1, 2, 3, 4, 6	Shared
Register Address: 1BH, 27	IA32_APIC_BASE		
APIC Location and Status (R/W) See Table 2-2. See Section 12.4.4, "Local APIC Status and Location."		0, 1, 2, 3, 4, 6	Unique
Register Address: 2AH, 42	MSR_EBC_HARD_POWERON		
Processor Hard Power-On Configuration (R/W) Enables and disables processor features. (R) Indicates current processor configuration.		0, 1, 2, 3, 4, 6	Shared
0	Output Tri-state Enabled (R) Indicates whether tri-state output is enabled (1) or disabled (0) as set by the strapping of SMI#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.		
1	Execute BIST (R) Indicates whether the execution of the BIST is enabled (1) or disabled (0) as set by the strapping of INIT#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.		
2	In Order Queue Depth (R) Indicates whether the in order queue depth for the system bus is 1 (1) or up to 12 (0) as set by the strapping of A7#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.		
3	MCERR# Observation Disabled (R) Indicates whether MCERR# observation is enabled (0) or disabled (1) as determined by the strapping of A9#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.		
4	BINIT# Observation Enabled (R) Indicates whether BINIT# observation is enabled (0) or disabled (1) as determined by the strapping of A10#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.		
6:5	APIC Cluster ID (R) Contains the logical APIC cluster ID value as set by the strapping of A12# and A11#. The logical cluster ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
7	Bus Park Disable (R) Indicates whether bus park is enabled (0) or disabled (1) as set by the strapping of A15#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.		
11:8	Reserved.		
13:12	Agent ID (R) Contains the logical agent ID value as set by the strapping of BR[3:0]. The logical ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.		
63:14	Reserved.		
Register Address: 2BH, 43	MSR_EBC_SOFT_POWERON		
Processor Soft Power-On Configuration (R/W) Enables and disables processor features.		0, 1, 2, 3, 4, 6	Shared
0	RCNT/SCNT On Request Encoding Enable (R/W) Controls the driving of RCNT/SCNT on the request encoding. Set to enable (1); clear to disabled (0, default).		
1	Data Error Checking Disable (R/W) Set to disable system data bus parity checking; clear to enable parity checking.		
2	Response Error Checking Disable (R/W) Set to disable (default); clear to enable.		
3	Address/Request Error Checking Disable (R/W) Set to disable (default); clear to enable.		
4	Initiator MCERR# Disable (R/W) Set to disable MCERR# driving for initiator bus requests (default); clear to enable.		
5	Internal MCERR# Disable (R/W) Set to disable MCERR# driving for initiator internal errors (default); clear to enable.		
6	BINIT# Driver Disable (R/W) Set to disable BINIT# driver (default); clear to enable driver.		
63:7	Reserved.		
Register Address: 2CH, 44	MSR_EBC_FREQUENCY_ID		
Processor Frequency Configuration The bit field layout of this MSR varies according to the MODEL value in the CPUID version information. The following bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding equal or greater than 2. (R) The field Indicates the current processor frequency configuration.		2,3, 4, 6	Shared
15:0	Reserved.		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	Bit Description	Model Availability	Shared/ Unique[1]
18:16	Scalable Bus Speed (R/W) Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz (Model 2) 000B 266 MHz (Model 3 or 4) 001B 133 MHz 010B 200 MHz 011B 166 MHz 100B 333 MHz (Model 6)		
	133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.		
	266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B and model encoding = 3 or 4. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B and model encoding = 6. All other values are reserved.		
23:19	Reserved.		
31:24	Core Clock Frequency to System Bus Frequency Ratio (R) The processor core clock frequency to system bus frequency ratio observed at the deassertion of the reset pin.		
63:32	Reserved.		
Register Address: 2CH, 44	MSR_EBC_FREQUENCY_ID		
Processor Frequency Configuration (R) The bit field layout of this MSR varies according to the MODEL value of the CPUID version information. This bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding less than 2. Indicates current processor frequency configuration.		0, 1	Shared
20:0	Reserved.		
23:21	Scalable Bus Speed (R/W) Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz All others values reserved.		
63:24	Reserved.		
Register Address: 3AH, 58	IA32_FEATURE_CONTROL		
Control Features in IA-32 Processor (R/W) See Table 2-2. (If CPUID.01H:ECX.[bit 5])		3, 4, 6	Unique
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
BIOS Update Trigger Register (W) See Table 2-2.		0, 1, 2, 3, 4, 6	Shared
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID		
BIOS Update Signature ID (R/W) See Table 2-2.		0, 1, 2, 3, 4, 6	Unique
Register Address: 9BH, 155	IA32_SMM_MONITOR_CTL		
SMM Monitor Configuration (R/W) See Table 2-2.		3, 4, 6	Unique
Register Address: FEH, 254	IA32_MTRRCAP		
MTRR Information See Section 13.11.1, “MTRR Feature Identification.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 174H, 372	IA32_SYSENTER_CS		
CS Register Target for CPL 0 Code (R/W) See Table 2-2 and Section 6.8.7, “Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 175H, 373	IA32_SYSENTER_ESP		
Stack Pointer for CPL 0 Stack (R/W) See Table 2-2 and Section 6.8.7, “Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 176H, 374	IA32_SYSENTER_EIP		
CPL 0 Code Entry Point (R/W) See Table 2-2 and Section 6.8.7, “Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 179H, 377	IA32_MCG_CAP		
Machine Check Capabilities (R) See Table 2-2 and Section 17.3.1.1, “IA32_MCG_CAP MSR.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 17AH, 378	IA32_MCG_STATUS		
Machine Check Status (R) See Table 2-2 and Section 17.3.1.2, “IA32_MCG_STATUS MSR.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 17BH, 379	IA32_MCG_CTL		
Machine Check Feature Enable (R/W) See Table 2-2 and Section 17.3.1.3, “IA32_MCG_CTL MSR.”			
Register Address: 180H, 384	MSR_MCG_RAX		
Machine Check EAX/RAX Save State See Section 17.3.2.6, “IA32_MCG Extended Machine Check State MSRs.”		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 181H, 385	MSR_MCG_RBX		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
Machine Check EBX/RBX Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 182H, 386	MSR_MCG_RCX		
Machine Check ECX/RCX Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 183H, 387	MSR_MCG_RDX		
Machine Check EDX/RDX Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 184H, 388	MSR_MCG_RSI		
Machine Check ESI/RSI Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 185H, 389	MSR_MCG_RDI		
Machine Check EDI/RDI Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 186H, 390	MSR_MCG_RBP		
Machine Check EBP/RBP Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 187H, 391	MSR_MCG_RSP		
Machine Check ESP/RSP Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 188H, 392	MSR_MCG_RFLAGS		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
Machine Check EFLAGS/RFLAG Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 189H, 393	MSR_MCG_RIP		
Machine Check EIP/RIP Save State See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.		
Register Address: 18AH, 394	MSR_MCG_MISC		
Machine Check Miscellaneous See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
0	DS When set, the bit indicates that a page assist or page fault occurred during DS normal operation. The processors response is to shut down. The bit is used as an aid for debugging DS handling code. It is the responsibility of the user (BIOS or operating system) to clear this bit for normal operation.		
63:1	Reserved.		
Register Address: 18BH—18FH, 395—399	MSR_MCG_RESERVED1—MSR_MCG_RESERVED5		
Reserved.			
Register Address: 190H, 400	MSR_MCG_R8		
Machine Check R8 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 191H, 401	MSR_MCG_R9		
Machine Check R9D/R9 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 192H, 402	MSR_MCG_R10		
Machine Check R10 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	Bit Description	Model Availability	Shared/ Unique[1]
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 193H, 403	MSR_MCG_R11		
Machine Check R11 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 194H, 404	MSR_MCG_R12		
Machine Check R12 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 195H, 405	MSR_MCG_R13		
Machine Check R13 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 196H, 406	MSR_MCG_R14		
Machine Check R14 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 197H, 407	MSR_MCG_R15		
Machine Check R15 See Section 17.3.2.6, "IA32_MCG Extended Machine Check State MSRs."		0, 1, 2, 3, 4, 6	Unique
63:0	Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.		
Register Address: 198H, 408	IA32_PERF_STATUS		
See Table 2-2. See Section 16.1, "Enhanced Intel Speedstep® Technology."		3, 4, 6	Unique
Register Address: 199H, 409	IA32_PERF_CTL		
See Table 2-2. See Section 16.1, "Enhanced Intel Speedstep® Technology."		3, 4, 6	Unique

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
Register Address: 19AH, 410	IA32_CLOCK_MODULATION		
Thermal Monitor Control (R/W) See Table 2-2 and Section 16.8.3, “Software Controlled Clock Modulation.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 19BH, 411	IA32_THERM_INTERRUPT		
Thermal Interrupt Control (R/W) See Section 16.8.2, “Thermal Monitor,” and Table 2-2.		0, 1, 2, 3, 4, 6	Unique
Register Address: 19CH, 412	IA32_THERM_STATUS		
Thermal Monitor Status (R/W) See Section 16.8.2, “Thermal Monitor,” and Table 2-2.		0, 1, 2, 3, 4, 6	Shared
Register Address: 19DH, 413	MSR_THERM2_CTL		
Thermal Monitor 2 Control			
For Family F, Model 3 processors: When read, specifies the value of the target TM2 transition last written. When set, it sets the next target value for TM2 transition.		3	Shared
For Family F, Model 4 and Model 6 processors: When read, specifies the value of the target TM2 transition last written. Writes may cause #GP exceptions.		4, 6	Shared
Register Address: 1A0H, 416	IA32_MISC_ENABLE		
Enable Miscellaneous Processor Features (R/W)		0, 1, 2, 3, 4, 6	Shared
0	Fast-Strings Enable. See Table 2-2.		
1	Reserved.		
2	x87 FPU Fopcode Compatibility Mode Enable		
3	Thermal Monitor 1 Enable See Section 16.8.2, “Thermal Monitor,” and Table 2-2.		
4	Split-Lock Disable When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios. When the bit is clear (default), normal split-locks are issued to the bus. This debug feature is specific to the Pentium 4 processor.		
5	Reserved.		
6	Third-Level Cache Disable (R/W) When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache. Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CR0, the page-level cache controls, and/or the MTRRs. See Section 13.5.4, “Disabling and Enabling the L3 Cache.”		
7	Performance Monitoring Available (R) See Table 2-2.		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
8	Suppress Lock Enable When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.		
9	Prefetch Queue Disable When set, disables the prefetch queue. When clear (default), enables the prefetch queue.		
10	FERR# Interrupt Reporting Enable (R/W) When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled. When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SMI#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt. This flag does not affect the normal operation of the FERR# pin (to indicate an unmasked floating-point error) when the STPCLK# pin is not asserted.		
11	Branch Trace Storage Unavailable (BTS_UNAVILABLE) (R) See Table 2-2. When set, the processor does not support branch trace storage (BTS); when clear, BTS is supported.		
12	PEBS_UNAVILABLE: Processor Event Based Sampling Unavailable (R) See Table 2-2. When set, the processor does not support processor event-based sampling (PEBS); when clear, PEBS is supported.		
13	TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.	3	
17:14	Reserved.		
18	ENABLE MONITOR FSM (R/W) See Table 2-2.	3, 4, 6	

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique**[1]
19	Adjacent Cache Line Prefetch Disable (R/W) When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector. Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.		
21:20	Reserved.		
22	Limit CPUID MAXVAL (R/W) See Table 2-2. Setting this can cause unexpected behavior to software that depends on the availability of CPUID leaves greater than 3.	3, 4, 6	
23	xTPR Message Disable (R/W) See Table 2-2.		Shared
24	L1 Data Cache Context Mode (R/W) When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 13.5.6, “L1 Data Cache Context Mode.” When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive. If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].		
33:25	Reserved.		
34	XD Bit Disable (R/W) See Table 2-3.		Unique
63:35	Reserved.		
Register Address: 1A1H, 417	MSR_PLATFORM_BRV		
Platform Feature Requirements (R)		3, 4, 6	Shared
17:0	Reserved.		
18	PLATFORM Requirements When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.		
63:19	Reserved.		
Register Address: 1D7H, 471	MSR_LER_FROM_LIP		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique**[1]
Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 19.13.3, "Last Exception Records."		0, 1, 2, 3, 4, 6	Unique
31:0	From Linear IP Linear address of the last branch instruction.		
63:32	Reserved.		
Register Address: 1D7H, 471	MSR_LER_FROM_LIP		
63:0	From Linear IP Linear address of the last branch instruction (If IA-32e mode is active).		Unique
Register Address: 1D8H, 472	MSR_LER_TO_LIP		
Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 19.13.3, "Last Exception Records."		0, 1, 2, 3, 4, 6	Unique
31:0	From Linear IP Linear address of the target of the last branch instruction.		
63:32	Reserved.		
Register Address: 1D8H, 472	MSR_LER_TO_LIP		
63:0	From Linear IP Linear address of the target of the last branch instruction (If IA-32e mode is active).		Unique
Register Address: 1D9H, 473	MSR_DEBUGCTLA		
Debug Control (R/W) Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 19.13.1, "MSR_DEBUGCTLA MSR."		0, 1, 2, 3, 4, 6	Unique
Register Address: 1DAH, 474	MSR_LASTBRANCH_TOS		
Last Branch Record Stack TOS (R/O) Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record). See Section 19.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture," and addresses 1DBH-1DEH and 680H-68FH.		0, 1, 2, 3, 4, 6	Unique
Register Address: 1DBH, 475	MSR_LASTBRANCH_0		
Last Branch Record 0 (R/O) One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took. MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH. See Section 19.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."		0, 1, 2	Unique

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique**[1]
Register Address: 1DCH, 476	MSR_LASTBRANCH_1		
Last Branch Record 1 See description of the MSR_LASTBRANCH_0 MSR at 1DBH.		0, 1, 2	Unique
Register Address: 1DDH, 477	MSR_LASTBRANCH_2		
Last Branch Record 2 See description of the MSR_LASTBRANCH_0 MSR at 1DBH.		0, 1, 2	Unique
Register Address: 1DEH, 478	MSR_LASTBRANCH_3		
Last Branch Record 3 See description of the MSR_LASTBRANCH_0 MSR at 1DBH.		0, 1, 2	Unique
Register Address: 200H, 512	IA32_MTRR_PHYSBASE0		
Variable Range Base MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 201H, 513	IA32_MTRR_PHYSMASK0		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 202H, 514	IA32_MTRR_PHYSBASE1		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 203H, 515	IA32_MTRR_PHYSMASK1		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 204H, 516	IA32_MTRR_PHYSBASE2		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 205H, 517	IA32_MTRR_PHYSMASK2		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs".		0, 1, 2, 3, 4, 6	Shared
Register Address: 206H, 518	IA32_MTRR_PHYSBASE3		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 207H, 519	IA32_MTRR_PHYSMASK3		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 208H, 520	IA32_MTRR_PHYSBASE4		
Variable Range Mask MTRR See Section 13.11.2.3, "Variable Range MTRRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 209H, 521	IA32_MTRR_PHYSMASK4		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
Variable Range Mask MTRR See Section 13.11.2.3, “Variable Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 20AH, 522	IA32_MTRR_PHYSBASE5		
Variable Range Mask MTRR See Section 13.11.2.3, “Variable Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 20BH, 523	IA32_MTRR_PHYSMASK5		
Variable Range Mask MTRR See Section 13.11.2.3, “Variable Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 20CH, 524	IA32_MTRR_PHYSBASE6		
Variable Range Mask MTRR See Section 13.11.2.3, “Variable Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 20DH, 525	IA32_MTRR_PHYSMASK6		
Variable Range Mask MTRR See Section 13.11.2.3, “Variable Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 20EH, 526	IA32_MTRR_PHYSBASE7		
Variable Range Mask MTRR See Section 13.11.2.3, “Variable Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 20FH, 527	IA32_MTRR_PHYSMASK7		
Variable Range Mask MTRR See Section 13.11.2.3, “Variable Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 250H, 592	IA32_MTRR_FIX64K_00000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 258H, 600	IA32_MTRR_FIX16K_80000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 259H, 601	IA32_MTRR_FIX16K_A0000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 268H, 616	IA32_MTRR_FIX4K_C0000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 269H, 617	IA32_MTRR_FIX4K_C8000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs”.		0, 1, 2, 3, 4, 6	Shared
Register Address: 26AH, 618	IA32_MTRR_FIX4K_D0000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs”.		0, 1, 2, 3, 4, 6	Shared

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	Bit Description	Model Availability	Shared/ Unique[1]
Register Address: 26BH, 619	IA32_MTRR_FIX4K_D8000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 26CH, 620	IA32_MTRR_FIX4K_E0000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 26DH, 621	IA32_MTRR_FIX4K_E8000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 26EH, 622	IA32_MTRR_FIX4K_F0000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 26FH, 623	IA32_MTRR_FIX4K_F8000		
Fixed Range MTRR See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 277H, 631	IA32_PAT		
Page Attribute Table See Section 13.11.2.2, “Fixed Range MTRRs.”		0, 1, 2, 3, 4, 6	Unique
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE		
Default Memory Types (R/W) See Table 2-2 and Section 13.11.2.1, “IA32_MTRR_DEF_TYPE MSR.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 300H, 768	MSR_BPU_COUNTER0		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 301H, 769	MSR_BPU_COUNTER1		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 302H, 770	MSR_BPU_COUNTER2		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 303H, 771	MSR_BPU_COUNTER3		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 304H, 772	MSR_MS_COUNTER0		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 305H, 773	MSR_MS_COUNTER1		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 306H, 774	MSR_MS_COUNTER2		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 307H, 775	MSR_MS_COUNTER3		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 308H, 776	MSR_FLAME_COUNTER0		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	Bit Description	Model Availability	Shared/ Unique[1]
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 309H, 777	MSR_FLAME_COUNTER1		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 30AH, 778	MSR_FLAME_COUNTER2		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 30BH, 779	MSR_FLAME_COUNTER3		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 30CH, 780	MSR_IQ_COUNTER0		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 30DH, 781	MSR_IQ_COUNTER1		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 30EH, 782	MSR_IQ_COUNTER2		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 30FH, 783	MSR_IQ_COUNTER3		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 310H, 784	MSR_IQ_COUNTER4		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 311H, 785	MSR_IQ_COUNTER5		
See Section 21.6.3.2, “Performance Counters.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 360H, 864	MSR_BPU_CCCR0		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 361H, 865	MSR_BPU_CCCR1		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 362H, 866	MSR_BPU_CCCR2		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 363H, 867	MSR_BPU_CCCR3		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 364H, 868	MSR_MS_CCCR0		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 365H, 869	MSR_MS_CCCR1		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 366H, 870	MSR_MS_CCCR2		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 367H, 871	MSR_MS_CCCR3		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 368H, 872	MSR_FLAME_CCCR0		
See Section 21.6.3.3, “CCCR MSRs.”		0, 1, 2, 3, 4, 6	Shared

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
Register Address: 369H, 873	MSR_FLAME_CCCR1		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 36AH, 874	MSR_FLAME_CCCR2		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 36BH, 875	MSR_FLAME_CCCR3		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 36CH, 876	MSR_IQ_CCCR0		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 36DH, 877	MSR_IQ_CCCR1		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 36EH, 878	MSR_IQ_CCCR2		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 36FH, 879	MSR_IQ_CCCR3		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 370H, 880	MSR_IQ_CCCR4		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 371H, 881	MSR_IQ_CCCR5		
See Section 21.6.3.3, "CCCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A0H, 928	MSR_BSU_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A1H, 929	MSR_BSU_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A2H, 930	MSR_FSB_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A3H, 931	MSR_FSB_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A4H, 932	MSR_FIRM_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A5H, 933	MSR_FIRM_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A6H, 934	MSR_FLAME_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A7H, 935	MSR_FLAME_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A8H, 936	MSR_DAC_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3A9H, 937	MSR_DAC_ESCR1		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3AAH, 938	MSR_MOB_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3ABH, 939	MSR_MOB_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3ACH, 940	MSR_PMH_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3ADH, 941	MSR_PMH_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3AEH, 942	MSR_SAAT_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3AFH, 943	MSR_SAAT_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B0H, 944	MSR_U2L_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B1H, 945	MSR_U2L_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B2H, 946	MSR_BPU_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B3H, 947	MSR_BPU_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B4H, 948	MSR_IS_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B5H, 949	MSR_IS_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B6H, 950	MSR_ITLB_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B7H, 951	MSR_ITLB_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B8H, 952	MSR_CRU_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3B9H, 953	MSR_CRU_ESCR1		
See Section 21.6.3.1, “ESCR MSRs.”		0, 1, 2, 3, 4, 6	Shared
Register Address: 3BAH, 954	MSR_IQ_ESCR0		
See Section 21.6.3.1, “ESCR MSRs.” This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.		0, 1, 2	Shared

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	Bit Description	Model Availability	Shared/ Unique[1]
Register Address: 3BBH, 955	MSR_IQ_ESCR1		
See Section 21.6.3.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.		0, 1, 2	Shared
Register Address: 3BCH, 956	MSR_RAT_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3BDH, 957	MSR_RAT_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3BEH, 958	MSR_SSU_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C0H, 960	MSR_MS_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C1H, 961	MSR_MS_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C2H, 962	MSR_TBPU_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C3H, 963	MSR_TBPU_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C4H, 964	MSR_TC_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C5H, 965	MSR_TC_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C8H, 968	MSR_IX_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3C9H, 969	MSR_IX_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3CAH, 970	MSR_ALF_ESCR0		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3CBH, 971	MSR_ALF_ESCR1		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3CCH, 972	MSR_CRU_ESCR2		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3CDH, 973	MSR_CRU_ESCR3		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3E0H, 992	MSR_CRU_ESCR4		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3E1H, 993	MSR_CRU_ESCR5		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3F0H, 1008	MSR_TC_PRECISE_EVENT		
See Section 21.6.3.1, "ESCR MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 3F1H, 1009	IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)		
Processor Event Based Sampling (PEBS) (R/W) Controls the enabling of processor event sampling and replay tagging.		0, 1, 2, 3, 4, 6	Shared
12:0	See https://perfmon-events.intel.com/.		
23:13	Reserved.		
24	UOP Tag Enables replay tagging when set.		
25	ENABLE_PEBS_MY_THR (R/W) Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 21.6.4.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is called ENABLE_PEBS in IA-32 processors that do not support Intel Hyper-Threading Technology.		
26	ENABLE_PEBS_OTH_THR (R/W) Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 21.6.4.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is reserved for IA-32 processors that do not support Intel Hyper-Threading Technology.		
63:27	Reserved.		
Register Address: 3F2H, 1010	MSR_PEBS_MATRIX_VERT		
See https://perfmon-events.intel.com/.		0, 1, 2, 3, 4, 6	Shared
Register Address: 400H, 1024	IA32_MC0_CTL		
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 401H, 1025	IA32_MC0_STATUS		
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		0, 1, 2, 3, 4, 6	Shared
Register Address: 402H, 1026	IA32_MC0_ADDR		
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		0, 1, 2, 3, 4, 6	Shared
Register Address: 403H, 1027	IA32_MC0_MISC		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." The IA32_MC0_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		0, 1, 2, 3, 4, 6	Shared
Register Address: 404H, 1028	IA32_MC1_CTL		
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 405H, 1029	IA32_MC1_STATUS		
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		0, 1, 2, 3, 4, 6	Shared
Register Address: 406H, 1030	IA32_MC1_ADDR		
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		0, 1, 2, 3, 4, 6	Shared
Register Address: 407H, 1031	IA32_MC1_MISC		
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." The IA32_MC1_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.			Shared
Register Address: 408H, 1032	IA32_MC2_CTL		
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 409H, 1033	IA32_MC2_STATUS		
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		0, 1, 2, 3, 4, 6	Shared
Register Address: 40AH, 1034	IA32_MC2_ADDR		
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.			
Register Address: 40BH, 1035	IA32_MC2_MISC		
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.			
Register Address: 40CH, 1036	IA32_MC3_CTL		
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 40DH, 1037	IA32_MC3_STATUS		
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		0, 1, 2, 3, 4, 6	Shared
Register Address: 40EH, 1038	IA32_MC3_ADDR		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		0, 1, 2, 3, 4, 6	Shared
Register Address: 40FH, 1039	IA32_MC3_MISC		
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." The IA32_MC3_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		0, 1, 2, 3, 4, 6	Shared
Register Address: 410H, 1040	IA32_MC4_CTL		
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		0, 1, 2, 3, 4, 6	Shared
Register Address: 411H, 1041	IA32_MC4_STATUS		
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		0, 1, 2, 3, 4, 6	Shared
Register Address: 412H, 1042	IA32_MC4_ADDR		
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.			
Register Address: 413H, 1043	IA32_MC4_MISC		
See Section 17.3.2.4, "IA32_MC**i**_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.			
Register Address: 480H, 1152	IA32_VMX_BASIC		
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2 and Appendix A.1, "Basic VMX Information."		3, 4, 6	Unique
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS		
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2 and Appendix A.3, "VM-Execution Controls."		3, 4, 6	Unique
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS		
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls," and Table 2-2.		3, 4, 6	Unique
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS		
Capability Reporting Register of VM-Exit Controls (R/O) See Appendix A.4, "VM-Exit Controls," and Table 2-2.		3, 4, 6	Unique
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	Bit Description	Model Availability	Shared/ Unique[1]
Capability Reporting Register of VM-Entry Controls (R/O) See Appendix A.5, "VM-Entry Controls," and Table 2-2.		3, 4, 6	Unique
Register Address: 485H, 1157	IA32_VMX_MISC		
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Appendix A.6, "Miscellaneous Data," and Table 2-2.		3, 4, 6	Unique
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0		
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Appendix A.7, "VMX-Fixed Bits in CR0," and Table 2-2.		3, 4, 6	Unique
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1		
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Appendix A.7, "VMX-Fixed Bits in CR0," and Table 2-2.		3, 4, 6	Unique
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0		
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Appendix A.8, "VMX-Fixed Bits in CR4," and Table 2-2.		3, 4, 6	Unique
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1		
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Appendix A.8, "VMX-Fixed Bits in CR4," and Table 2-2.		3, 4, 6	Unique
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM		
Capability Reporting Register of VMCS Field Enumeration (R/O) See Appendix A.9, "VMCS Enumeration," and Table 2-2.		3, 4, 6	Unique
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2		
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, "VM-Execution Controls," and Table 2-2.		3, 4, 6	Unique
Register Address: 600H, 1536	IA32_DS_AREA		
DS Save Area (R/W) See Table 2-2 and Section 21.6.3.4, "Debug Store (DS) Mechanism."		0, 1, 2, 3, 4, 6	Unique
Register Address: 680H, 1664	MSR_LASTBRANCH_0_FROM_IP		
Last Branch Record 0 (R/W) One of 16 pairs of last branch record registers on the last branch record stack (680H-68FH). This part of the stack contains pointers to the source instruction for one of the last 16 branches, exceptions, or interrupts taken by the processor. The MSRs at 680H-68FH, 6C0H-6CfH are not available in processor releases before family 0FH, model 03H. These MSRs replace MSRs previously located at 1DBH-1DEH. which performed the same function for early releases. See Section 19.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."		3, 4, 6	Unique
Register Address: 681H, 1665	MSR_LASTBRANCH_1_FROM_IP		
Last Branch Record 1 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 682H, 1666	MSR_LASTBRANCH_2_FROM_IP		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
Last Branch Record 2 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 683H, 1667	MSR_LASTBRANCH_3_FROM_IP		
Last Branch Record 3 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 684H, 1668	MSR_LASTBRANCH_4_FROM_IP		
Last Branch Record 4 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 685H, 1669	MSR_LASTBRANCH_5_FROM_IP		
Last Branch Record 5 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 686H, 1670	MSR_LASTBRANCH_6_FROM_IP		
Last Branch Record 6 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 687H, 1671	MSR_LASTBRANCH_7_FROM_IP		
Last Branch Record 7 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 688H, 1672	MSR_LASTBRANCH_8_FROM_IP		
Last Branch Record 8 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 689H, 1673	MSR_LASTBRANCH_9_FROM_IP		
Last Branch Record 9 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 68AH, 1674	MSR_LASTBRANCH_10_FROM_IP		
Last Branch Record 10 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 68BH, 1675	MSR_LASTBRANCH_11_FROM_IP		
Last Branch Record 11 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 68CH, 1676	MSR_LASTBRANCH_12_FROM_IP		
Last Branch Record 12 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 68DH, 1677	MSR_LASTBRANCH_13_FROM_IP		
Last Branch Record 13 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 68EH, 1678	MSR_LASTBRANCH_14_FROM_IP		
Last Branch Record 14 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique**[1]
Register Address: 68FH, 1679	MSR_LASTBRANCH_15_FROM_IP		
Last Branch Record 15 See description of MSR_LASTBRANCH_0 at 680H.		3, 4, 6	Unique
Register Address: 6C0H, 1728	MSR_LASTBRANCH_0_TO_IP		
Last Branch Record 0 (R/W) One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the destination instruction for one of the last 16 branches, exceptions, or interrupts that the processor took. See Section 19.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture.”		3, 4, 6	Unique
Register Address: 6C1H, 1729	MSR_LASTBRANCH_1_TO_IP		
Last Branch Record 1 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C2H, 1730	MSR_LASTBRANCH_2_TO_IP		
Last Branch Record 2 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C3H, 1731	MSR_LASTBRANCH_3_TO_IP		
Last Branch Record 3 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C4H, 1732	MSR_LASTBRANCH_4_TO_IP		
Last Branch Record 4 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C5H, 1733	MSR_LASTBRANCH_5_TO_IP		
Last Branch Record 5 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C6H, 1734	MSR_LASTBRANCH_6_TO_IP		
Last Branch Record 6 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C7H, 1735	MSR_LASTBRANCH_7_TO_IP		
Last Branch Record 7 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C8H, 1736	MSR_LASTBRANCH_8_TO_IP		
Last Branch Record 8 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6C9H, 1737	MSR_LASTBRANCH_9_TO_IP		
Last Branch Record 9 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6CAH, 1738	MSR_LASTBRANCH_10_TO_IP		

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	**Bit Description**	**Model Availability**	**Shared/ Unique[1]**
Last Branch Record 10 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6CBH, 1739	MSR_LASTBRANCH_11_TO_IP		
Last Branch Record 11 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6CCH, 1740	MSR_LASTBRANCH_12_TO_IP		
Last Branch Record 12 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6CDH, 1741	MSR_LASTBRANCH_13_TO_IP		
Last Branch Record 13 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6CEH, 1742	MSR_LASTBRANCH_14_TO_IP		
Last Branch Record 14 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: 6CFH, 1743	MSR_LASTBRANCH_15_TO_IP		
Last Branch Record 15 See description of MSR_LASTBRANCH_0 at 6C0H.		3, 4, 6	Unique
Register Address: C000_0080H	IA32_EFER		
Extended Feature Enables See Table 2-2.		3, 4, 6	Unique
Register Address: C000_0081H	IA32_STAR		
System Call Target Address (R/W) See Table 2-2.		3, 4, 6	Unique
Register Address: C000_0082H	IA32_LSTAR		
IA-32e Mode System Call Target Address (R/W) See Table 2-2.		3, 4, 6	Unique
Register Address: C000_0084H	IA32_FMASK		
System Call Flag Mask (R/W) See Table 2-2.		3, 4, 6	Unique
Register Address: C000_0100H	IA32_FS_BASE		
Map of BASE Address of FS (R/W) See Table 2-2.		3, 4, 6	Unique
Register Address: C000_0101H	IA32_GS_BASE		
Map of BASE Address of GS (R/W) See Table 2-2.		3, 4, 6	Unique
Register Address: C000_0102H	IA32_KERNEL_GS_BASE		
Swap Target of BASE Address of GS (R/W) See Table 2-2.		3, 4, 6	Unique

Table 2-63. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address: Hex, Decimal	Register Name		
Register Information / Bit Fields	Bit Description	Model Availability	Shared/ Unique[1]
NOTES 1. For HT-enabled processors, there may be more than one logical processors per physical unit. If an MSR is Shared, this means that one MSR is shared between logical processors. If an MSR is unique, this means that each logical processor has its own MSR.			

2.19.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache

The MSRs listed in Table 2-64 apply to Intel® Xeon® Processor MP with up to 8MB level three cache. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 3 or 4 (see CPUID instruction for more details).

Table 2-64. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache

Register Address: Hex	Register Name		
Register Information		Model Availability	Shared/ Unique
Register Address: 107CCH	MSR_IFSB_BUSQ0		
IFSB BUSQ Event Control and Counter Register (R/W) See Section 21.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”		3, 4	Shared
Register Address: 107CDH	MSR_IFSB_BUSQ1		
IFSB BUSQ Event Control and Counter Register (R/W)		3, 4	Shared
Register Address: 107CEH	MSR_IFSB_SNPQ0		
IFSB SNPQ Event Control and Counter Register (R/W) See Section 21.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”		3, 4	Shared
Register Address: 107CFH	MSR_IFSB_SNPQ1		
IFSB SNPQ Event Control and Counter Register (R/W)		3, 4	Shared
Register Address: 107D0H	MSR_EFSB_DRDY0		
EFSB DRDY Event Control and Counter Register (R/W) See Section 21.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”		3, 4	Shared
Register Address: 107D1H	MSR_EFSB_DRDY1		
EFSB DRDY Event Control and Counter Register (R/W)		3, 4	Shared
Register Address: 107D2H	MSR_IFSB_CTL6		
IFSB Latency Event Control Register (R/W) See Section 21.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”		3, 4	Shared
Register Address: 107D3H	MSR_IFSB_CNTR7		
IFSB Latency Event Counter Register (R/W) See Section 21.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”		3, 4	Shared

The MSRs listed in Table 2-65 apply to Intel® Xeon® Processor 7100 series. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 6 (See CPUID instruction for more details.). The performance monitoring MSRs listed in Table 2-65 are shared between logical processors in the same core, but are replicated for each core.

Table 2-65. MSRs Unique to Intel® Xeon® Processor 7100 Series

Register Address: Hex	Register Name		
Register Information		**Model Availability**	**Shared/ Unique**
Register Address: 107CCH	MSR_EMON_L3_CTR_CTL0		
GBUSQ Event Control and Counter Register (R/W) See Section 21.6.6, "Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache."		6	Shared
Register Address: 107CDH	MSR_EMON_L3_CTR_CTL1		
GBUSQ Event Control and Counter Register (R/W)		6	Shared
Register Address: 107CEH	MSR_EMON_L3_CTR_CTL2		
GSNPQ Event Control and Counter Register (R/W) See Section 21.6.6, "Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache."		6	Shared
Register Address: 107CFH	MSR_EMON_L3_CTR_CTL3		
GSNPQ Event Control and Counter Register (R/W)		6	Shared
Register Address: 107D0H	MSR_EMON_L3_CTR_CTL4		
FSB Event Control and Counter Register (R/W) See Section 21.6.6, "Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache."		6	Shared
Register Address: 107D1H	MSR_EMON_L3_CTR_CTL5		
FSB Event Control and Counter Register (R/W)		6	Shared
Register Address: 107D2H	MSR_EMON_L3_CTR_CTL6		
FSB Event Control and Counter Register (R/W)		6	Shared
Register Address: 107D3H	MSR_EMON_L3_CTR_CTL7		
FSB Event Control and Counter Register (R/W)		6	Shared

2.20 MSRS IN INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Model-specific registers (MSRs) for Intel Core Solo, Intel Core Duo processors, and Dual-core Intel Xeon processor LV are listed in Table 2-66. The column "Shared/Unique" applies to Intel Core Duo processor. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 0H, 0	P5_MC_ADDR	
See Section 2.23, "MSRs in Pentium Processors," and Table 2-2.		Unique
Register Address: 1H, 1	P5_MC_TYPE	
See Section 2.23, "MSRs in Pentium Processors," and Table 2-2.		Unique

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 10.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.		Unique
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER	
See Section 19.17, "Time-Stamp Counter," and Table 2-2.		Unique
Register Address: 17H, 23	IA32_PLATFORM_ID	
Platform ID (R) See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.		Shared
Register Address: 1BH, 27	IA32_APIC_BASE	
See Section 12.4.4, "Local APIC Status and Location," and Table 2-2.		Unique
Register Address: 2AH, 42	MSR_EBL_CR_POWERON	
Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.		Shared
0	Reserved.	
1	Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.	
2	Response Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.	
3	MCERR# Drive Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.	
4	Address Parity Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.	
6: 5	Reserved.	
7	BINIT# Driver Enable (R/W) 1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.	
8	Output Tri-state Enabled (R/O) 1 = Enabled; 0 = Disabled.	
9	Execute BIST (R/O) 1 = Enabled; 0 = Disabled.	
10	MCERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled.	
11	Reserved.	
12	BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled.	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
13	Reserved	
14	1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes	
15	Reserved.	
17:16	APIC Cluster ID (R/O)	
18	System Bus Frequency (R/O) 0 = 100 MHz. 1 = Reserved.	
19	Reserved.	
21: 20	Symmetric Arbitration ID (R/O)	
26:22	Clock Frequency Ratio (R/O)	
Register Address: 3AH, 58	IA32_FEATURE_CONTROL	
Control Features in IA-32 Processor (R/W) See Table 2-2.		Unique
Register Address: 40H, 64	MSR_LASTBRANCH_0	
Last Branch Record 0 (R/W) One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the ‘from’ address and bits 63-32 hold the ‘to’ address. See also: • Last Branch Record Stack TOS at 1C9H. • Section 19.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors).”		Unique
Register Address: 41H, 65	MSR_LASTBRANCH_1	
Last Branch Record 1 (R/W) See description of MSR_LASTBRANCH_0.		Unique
Register Address: 42H, 66	MSR_LASTBRANCH_2	
Last Branch Record 2 (R/W) See description of MSR_LASTBRANCH_0.		Unique
Register Address: 43H, 67	MSR_LASTBRANCH_3	
Last Branch Record 3 (R/W) See description of MSR_LASTBRANCH_0.		Unique
Register Address: 44H, 68	MSR_LASTBRANCH_4	
Last Branch Record 4 (R/W) See description of MSR_LASTBRANCH_0.		Unique
Register Address: 45H, 69	MSR_LASTBRANCH_5	
Last Branch Record 5 (R/W) See description of MSR_LASTBRANCH_0.		Unique
Register Address: 46H, 70	MSR_LASTBRANCH_6	
Last Branch Record 6 (R/W) See description of MSR_LASTBRANCH_0.		Unique
Register Address: 47H, 71	MSR_LASTBRANCH_7	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Last Branch Record 7 (R/W) See description of MSR_LASTBRANCH_0.		Unique
Register Address: 79H, 121	IA32_BIOS_UPDT_TRIG	
BIOS Update Trigger Register (W) See Table 2-2.		Unique
Register Address: 8BH, 139	IA32_BIOS_SIGN_ID	
BIOS Update Signature ID (R/W) See Table 2-2.		Unique
Register Address: C1H, 193	IA32_PMC0	
Performance Counter Register See Table 2-2.		Unique
Register Address: C2H, 194	IA32_PMC1	
Performance Counter Register See Table 2-2.		Unique
Register Address: CDH, 205	MSR_FSB_FREQ	
Scaleable Bus Speed (R/O) This field indicates the scalable bus clock speed.		Shared
2:0	▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 101B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.	
63:3	Reserved.	
Register Address: E7H, 231	IA32_MPERF	
Maximum Performance Frequency Clock Count (R/W) See Table 2-2.		Unique
Register Address: E8H, 232	IA32_APERF	
Actual Performance Frequency Clock Count (R/W) See Table 2-2.		Unique
Register Address: FEH, 254	IA32_MTRRCAP	
See Table 2-2.		Unique
Register Address: 11EH, 281	MSR_BBL_CR_CTL3	
Control Register 3 Used to configure the L2 Cache.		Shared
0	L2 Hardware Enabled (R/O) 1 = If the L2 is hardware-enabled. 0 = Indicates if the L2 is hardware-disabled.	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
7:1	Reserved.	
8	L2 Enabled (R/W) 1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.	
22:9	Reserved.	
23	L2 Not Present (R/O) 0 = L2 Present. 1 = L2 Not Present.	
63:24	Reserved.	
Register Address: 174H, 372	IA32_SYSENTER_CS	
See Table 2-2.		Unique
Register Address: 175H, 373	IA32_SYSENTER_ESP	
See Table 2-2.		Unique
Register Address: 176H, 374	IA32_SYSENTER_EIP	
See Table 2-2.		Unique
Register Address: 179H, 377	IA32_MCG_CAP	
See Table 2-2.		Unique
Register Address: 17AH, 378	IA32_MCG_STATUS	
Global Machine Check Status		Unique
0	RIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.	
1	EIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.	
2	MCIP When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.	
63:3	Reserved	
Register Address: 186H, 390	IA32_PERFEVTSEL0	
See Table 2-2.		Unique
Register Address: 187H, 391	IA32_PERFEVTSEL1	
See Table 2-2.		Unique
Register Address: 198H, 408	IA32_PERF_STATUS	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
See Table 2-2.		Shared
Register Address: 199H, 409	IA32_PERF_CTL	
See Table 2-2.		Unique
Register Address: 19AH, 410	IA32_CLOCK_MODULATION	
Clock Modulation (R/W) See Table 2-2.		Unique
Register Address: 19BH, 411	IA32_THERM_INTERRUPT	
Thermal Interrupt Control (R/W) See Table 2-2 and Section 16.8.2, “Thermal Monitor.”		Unique
Register Address: 19CH, 412	IA32_THERM_STATUS	
Thermal Monitor Status (R/W) See Table 2-2 and Section 16.8.2, “Thermal Monitor”.		Unique
Register Address: 19DH, 413	MSR_THERM2_CTL	
Thermal Monitor 2 Control		Unique
15:0	Reserved.	
16	TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.	
63:16	Reserved.	
Register Address: 1A0H, 416	IA32_MISC_ENABLE	
Enable Miscellaneous Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.		
2:0	Reserved.	
3	Automatic Thermal Control Circuit Enable (R/W) See Table 2-2.	Unique
6:4	Reserved.	
7	Performance Monitoring Available (R) See Table 2-2.	Shared
9:8	Reserved.	
10	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.	Shared
11	Branch Trace Storage Unavailable (R/O) See Table 2-2.	Shared

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
12	Reserved.	
13	TM2 Enable (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.	Shared
15:14	Reserved.	
16	Enhanced Intel SpeedStep Technology Enable (R/W) 1 = Enhanced Intel SpeedStep Technology enabled	Shared
18	ENABLE MONITOR FSM (R/W) See Table 2-2.	Shared
19	Reserved.	
22	Limit CPUID Maxval (R/W) See Table 2-2. Setting this bit may cause behavior in software that depends on the availability of CPUID leaves greater than 2.	Shared
33:23	Reserved.	
34	XD Bit Disable (R/W) See Table 2-3.	Shared
63:35	Reserved.	
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS	
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).		Unique
Register Address: 1D9H, 473	IA32_DEBUGCTL	
Debug Control (R/W) Controls how several debug features are used. Bit definitions are discussed in Table 2-2.		Unique
Register Address: 1DDH, 477	MSR_LER_FROM_LIP	
Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Unique
Register Address: 1DEH, 478	MSR_LER_TO_LIP	
Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.		Unique
Register Address: 200H, 512	MTRRphysBase0	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Memory Type Range Registers		Unique
Register Address: 201H, 513	MTRRphysMask0	
Memory Type Range Registers		Unique
Register Address: 202H, 514	MTRRphysBase1	
Memory Type Range Registers		Unique
Register Address: 203H, 515	MTRRphysMask1	
Memory Type Range Registers		Unique
Register Address: 204H, 516	MTRRphysBase2	
Memory Type Range Registers		Unique
Register Address: 205H, 517	MTRRphysMask2	
Memory Type Range Registers		Unique
Register Address: 206H, 518	MTRRphysBase3	
Memory Type Range Registers		Unique
Register Address: 207H, 519	MTRRphysMask3	
Memory Type Range Registers		Unique
Register Address: 208H, 520	MTRRphysBase4	
Memory Type Range Registers		Unique
Register Address: 209H, 521	MTRRphysMask4	
Memory Type Range Registers		Unique
Register Address: 20AH, 522	MTRRphysBase5	
Memory Type Range Registers		Unique
Register Address: 20BH, 523	MTRRphysMask5	
Memory Type Range Registers		Unique
Register Address: 20CH, 524	MTRRphysBase6	
Memory Type Range Registers		Unique
Register Address: 20DH, 525	MTRRphysMask6	
Memory Type Range Registers		Unique
Register Address: 20EH, 526	MTRRphysBase7	
Memory Type Range Registers		Unique
Register Address: 20FH, 527	MTRRphysMask7	
Memory Type Range Registers		Unique
Register Address: 250H, 592	MTRRfix64K_00000	
Memory Type Range Registers		Unique
Register Address: 258H, 600	MTRRfix16K_80000	
Memory Type Range Registers		Unique
Register Address: 259H, 601	MTRRfix16K_A0000	
Memory Type Range Registers		Unique

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Register Address: 268H, 616	MTRRfix4K_C0000	
Memory Type Range Registers		Unique
Register Address: 269H, 617	MTRRfix4K_C8000	
Memory Type Range Registers		Unique
Register Address: 26AH, 618	MTRRfix4K_D0000	
Memory Type Range Registers		Unique
Register Address: 26BH, 619	MTRRfix4K_D8000	
Memory Type Range Registers		Unique
Register Address: 26CH, 620	MTRRfix4K_E0000	
Memory Type Range Registers		Unique
Register Address: 26DH, 621	MTRRfix4K_E8000	
Memory Type Range Registers		Unique
Register Address: 26EH, 622	MTRRfix4K_F0000	
Memory Type Range Registers		Unique
Register Address: 26FH, 623	MTRRfix4K_F8000	
Memory Type Range Registers		Unique
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE	
Default Memory Types (R/W) See Table 2-2 and Section 13.11.2.1, “IA32_MTRR_DEF_TYPE MSR.”		Unique
Register Address: 400H, 1024	IA32_MC0_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Unique
Register Address: 401H, 1025	IA32_MC0_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS.”		Unique
Register Address: 402H, 1026	IA32_MC0_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 404H, 1028	IA32_MC1_CTL	
See Section 17.3.2.1, “IA32_MC**i**_CTL MSRs.”		Unique
Register Address: 405H, 1029	IA32_MC1_STATUS	
See Section 17.3.2.2, “IA32_MCi_STATUS MSRS.”		Unique
Register Address: 406H, 1030	IA32_MC1_ADDR	
See Section 17.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 408H, 1032	IA32_MC2_CTL	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Unique
Register Address: 409H, 1033	IA32_MC2_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Unique
Register Address: 40AH, 1034	IA32_MC2_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 40CH, 1036	MSR_MC4_CTL	
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."		Unique
Register Address: 40DH, 1037	MSR_MC4_STATUS	
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."		Unique
Register Address: 40EH, 1038	MSR_MC4_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 410H, 1040	IA32_MC3_CTL	
IA32_MC3_CTL	See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."	
Register Address: 411H, 1041	IA32_MC3_STATUS	
IA32_MC3_STATUS	See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."	
Register Address: 412H, 1042	MSR_MC3_ADDR	
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.		Unique
Register Address: 413H, 1043	MSR_MC3_MISC	
Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.		Unique
Register Address: 414H, 1044	MSR_MC5_CTL	
Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).		Unique
Register Address: 415H, 1045	MSR_MC5_STATUS	
Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.		Unique
Register Address: 416H, 1046	MSR_MC5_ADDR	
Machine Check Error Reporting Register - contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MCi_STATUS register is set.		Unique
Register Address: 417H, 1047	MSR_MC5_MISC	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.		Unique
Register Address: 480H, 1152	IA32_VMX_BASIC	
Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2 and Appendix A.1, “Basic VMX Information.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 481H, 1153	IA32_VMX_PINBASED_CTLS	
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Appendix A.3, “VM-Execution Controls.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 482H, 1154	IA32_VMX_PROCBASED_CTLS	
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, “VM-Execution Controls.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 483H, 1155	IA32_VMX_EXIT_CTLS	
Capability Reporting Register of VM-Exit Controls (R/O) See Appendix A.4, “VM-Exit Controls.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 484H, 1156	IA32_VMX_ENTRY_CTLS	
Capability Reporting Register of VM-Entry Controls (R/O) See Appendix A.5, “VM-Entry Controls.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 485H, 1157	IA32_VMX_MISC	
Reporting Register of Miscellaneous VMX Capabilities (R/O) See Appendix A.6, “Miscellaneous Data.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 486H, 1158	IA32_VMX_CR0_FIXED0	
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Appendix A.7, “VMX-Fixed Bits in CR0.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 487H, 1159	IA32_VMX_CR0_FIXED1	
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Appendix A.7, “VMX-Fixed Bits in CR0.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 488H, 1160	IA32_VMX_CR4_FIXED0	
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Appendix A.8, “VMX-Fixed Bits in CR4.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 489H, 1161	IA32_VMX_CR4_FIXED1	
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Appendix A.8, “VMX-Fixed Bits in CR4.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 48AH, 1162	IA32_VMX_VMCS_ENUM	
Capability Reporting Register of VMCS Field Enumeration (R/O) See Appendix A.9, “VMCS Enumeration.” (If CPUID.01H:ECX.[bit 5])		Unique
Register Address: 48BH, 1163	IA32_VMX_PROCBASED_CTLS2	
Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, “VM-Execution Controls.” (If CPUID.01H:ECX.[bit 5] and IA32_VMX_PROCBASED_CTLS[bit 63])		Unique
Register Address: 600H, 1536	IA32_DS_AREA	

Table 2-66. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address: Hex, Decimal	Register Name	
Register Information / Bit Fields	**Bit Description**	**Shared/ Unique**
DS Save Area (R/W) See Table 2-2 and Section 21.6.3.4, "Debug Store (DS) Mechanism."		Unique
31:0	DS Buffer Management Area Linear address of the first byte of the DS buffer management area.	
63:32	Reserved.	
Register Address: C000_0080H	IA32_EFER	
See Table 2-2.		Unique
10:0	Reserved.	
11	Execute Disable Bit Enable	
63:12	Reserved.	

2.21 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 2.22 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

Table 2-67. MSRs in Pentium M Processors

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Register Address: 0H, 0	P5_MC_ADDR
See Section 2.23, "MSRs in Pentium Processors."	
Register Address: 1H, 1	P5_MC_TYPE
See Section 2.23, "MSRs in Pentium Processors."	
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER
See Section 19.17, "Time-Stamp Counter," and see Table 2-2.	
Register Address: 17H, 23	IA32_PLATFORM_ID
Platform ID (R) See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.	
Register Address: 2AH, 42	MSR_EBL_CR_POWERON
Processor Hard Power-On Configuration (R/W) Enables and disables processor features. (R) Indicates current processor configuration.	
0	Reserved.
1	Data Error Checking Enable (R) 0 = Disabled. Always 0 on the Pentium M processor.

Table 2-67. MSRs in Pentium M Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
2	Response Error Checking Enable (R) 0 = Disabled. Always 0 on the Pentium M processor.
3	MCERR# Drive Enable (R) 0 = Disabled. Always 0 on the Pentium M processor.
4	Address Parity Enable (R) 0 = Disabled. Always 0 on the Pentium M processor.
6:5	Reserved.
7	BINIT# Driver Enable (R) 1 = Enabled; 0 = Disabled. Always 0 on the Pentium M processor.
8	Output Tri-state Enabled (R/O) 1 = Enabled; 0 = Disabled.
9	Execute BIST (R/O) 1 = Enabled; 0 = Disabled.
10	MCERR# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled. Always 0 on the Pentium M processor.
11	Reserved.
12	BINIT# Observation Enabled (R/O) 1 = Enabled; 0 = Disabled. Always 0 on the Pentium M processor.
13	Reserved.
14	1 MByte Power on Reset Vector (R/O) 1 = 1 MByte; 0 = 4 GBytes. Always 0 on the Pentium M processor.
15	Reserved.
17:16	APIC Cluster ID (R/O) Always 00B on the Pentium M processor.
18	System Bus Frequency (R/O) 0 = 100 MHz. 1 = Reserved. Always 0 on the Pentium M processor.
19	Reserved.
21: 20	Symmetric Arbitration ID (R/O) Always 00B on the Pentium M processor.
26:22	Clock Frequency Ratio (R/O)
Register Address: 40H, 64	MSR_LASTBRANCH_0

Table 2-67. MSRs in Pentium M Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Last Branch Record 0 (R/W) One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the to address. See also: ▪ Last Branch Record Stack TOS at 1C9H. ▪ Section 19.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."	
Register Address: 41H, 65	MSR_LASTBRANCH_1
Last Branch Record 1 (R/W) See description of MSR_LASTBRANCH_0.	
Register Address: 42H, 66	MSR_LASTBRANCH_2
Last Branch Record 2 (R/W) See description of MSR_LASTBRANCH_0.	
Register Address: 43H, 67	MSR_LASTBRANCH_3
Last Branch Record 3 (R/W) See description of MSR_LASTBRANCH_0.	
Register Address: 44H, 68	MSR_LASTBRANCH_4
Last Branch Record 4 (R/W) See description of MSR_LASTBRANCH_0.	
Register Address: 45H, 69	MSR_LASTBRANCH_5
Last Branch Record 5 (R/W) See description of MSR_LASTBRANCH_0.	
Register Address: 46H, 70	MSR_LASTBRANCH_6
Last Branch Record 6 (R/W) See description of MSR_LASTBRANCH_0.	
Register Address: 47H, 71	MSR_LASTBRANCH_7
Last Branch Record 7 (R/W) See description of MSR_LASTBRANCH_0.	
Register Address: 119H, 281	MSR_BBL_CR_CTL
Control Register Used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response.	
63:0	Reserved.
Register Address: 11EH, 281	MSR_BBL_CR_CTL3
Control Register 3 Used to configure the L2 Cache.	
0	L2 Hardware Enabled (R/O) 1 = If the L2 is hardware-enabled. 0 = Indicates if the L2 is hardware-disabled.
4:1	Reserved.

Table 2-67. MSRs in Pentium M Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
5	ECC Check Enable (R/O) This bit enables ECC checking on the cache data bus. ECC is always generated on write cycles. 0 = Disabled (default). 1 = Enabled. For the Pentium M processor, ECC checking on the cache data bus is always enabled.
7:6	Reserved.
8	L2 Enabled (R/W) 1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
22:9	Reserved.
23	L2 Not Present (R/O) 0 = L2 Present. 1 = L2 Not Present.
63:24	Reserved.
Register Address: 179H, 377	IA32_MCG_CAP
Read-only register that provides information about the machine-check architecture of the processor.	
7:0	Count (R/O) Indicates the number of hardware unit error reporting banks available in the processor.
8	IA32_MCG_CTL Present (R/O) 1 = Indicates that the processor implements the MSR_MCG_CTL register found at MSR 17BH. 0 = Not supported.
63:9	Reserved.
Register Address: 17AH, 378	IA32_MCG_STATUS
Global Machine Check Status	
0	RIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.
1	EIPV When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
2	MCIP When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
63:3	Reserved.
Register Address: 198H, 408	IA32_PERF_STATUS

Table 2-67. MSRs in Pentium M Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
See Table 2-2.	
Register Address: 199H, 409	IA32_PERF_CTL
See Table 2-2.	
Register Address: 19AH, 410	IA32_CLOCK_MODULATION
Clock Modulation (R/W). See Table 2-2 and Section 16.8.3, “Software Controlled Clock Modulation.”	
Register Address: 19BH, 411	IA32_THERM_INTERRUPT
Thermal Interrupt Control (R/W) See Table 2-2 and Section 16.8.2, “Thermal Monitor.”	
Register Address: 19CH, 412	IA32_THERM_STATUS
Thermal Monitor Status (R/W) See Table 2-2 and Section 16.8.2, “Thermal Monitor.”	
Register Address: 19DH, 413	MSR_THERM2_CTL
Thermal Monitor 2 Control	
15:0	Reserved.
16	TM_SELECT (R/W) Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
63:16	Reserved.
Register Address: 1A0H, 416	IA32_MISC_ENABLE
Enable Miscellaneous Processor Features (R/W) Allows a variety of processor functions to be enabled and disabled.	
2:0	Reserved.
3	Automatic Thermal Control Circuit Enable (R/W) 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows processor clocks to be automatically modulated based on the processor’s thermal sensor operation. 0 = Disabled (default). The automatic thermal control circuit enable bit determines if the thermal control circuit (TCC) will be activated when the processor’s internal thermal sensor determines the processor is about to exceed its maximum operating temperature. When the TCC is activated and TM1 is enabled, the processors clocks will be forced to a 50% duty cycle. BIOS must enable this feature. The bit should not be confused with the on-demand thermal control circuit enable bit.
6:4	Reserved.
7	Performance Monitoring Available (R) 1 = Performance monitoring enabled. 0 = Performance monitoring disabled.

Table 2-67. MSRs in Pentium M Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
9:8	Reserved.
10	FERR# Multiplexing Enable (R/W) 1 = FERR# asserted by the processor to indicate a pending break event within the processor. 0 = Indicates compatible FERR# signaling behavior. This bit must be set to 1 to support XAPIC interrupt model usage.
	Branch Trace Storage Unavailable (R/O) 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported
12	Processor Event Based Sampling Unavailable (R/O) 1 = Processor does not support processor event based sampling (PEBS); 0 = PEBS is supported. The Pentium M processor does not support PEBS.
15:13	Reserved.
16	Enhanced Intel SpeedStep Technology Enable (R/W) 1 = Enhanced Intel SpeedStep Technology enabled. On the Pentium M processor, this bit may be configured to be read-only.
22:17	Reserved.
23	xTPR Message Disable (R/W) When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority. The default is processor specific.
63:24	Reserved.
Register Address: 1C9H, 457	MSR_LASTBRANCH_TOS
Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: ▪ MSR_LASTBRANCH_0_FROM_IP (at 40H). ▪ Section 19.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."	
Register Address: 1D9H, 473	MSR_DEBUGCTLB
Debug Control (R/W) Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 19.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."	
Register Address: 1DDH, 477	MSR_LER_TO_LIP
Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 19.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)," and Section 19.16.2, "Last Branch and Last Exception MSRs."	
Register Address: 1DEH, 478	MSR_LER_FROM_LIP

Table 2-67. MSRs in Pentium M Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Last Exception Record From Linear IP (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 19.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)," and Section 19.16.2, "Last Branch and Last Exception MSRs."	
Register Address: 2FFH, 767	IA32_MTRR_DEF_TYPE
Default Memory Types (R/W) Sets the memory type for the regions of physical memory that are not mapped by the MTRRs. See Section 13.11.2.1, "IA32_MTRR_DEF_TYPE MSR."	
Register Address: 400H, 1024	IA32_MC0_CTL
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."	
Register Address: 401H, 1025	IA32_MC0_STATUS
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."	
Register Address: 402H, 1026	IA32_MC0_ADDR
See Section 14.3.2.3., "IA32_MCi_ADDR MSRs". The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.	
Register Address: 404H, 1028	IA32_MC1_CTL
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."	
Register Address: 405H, 1029	IA32_MC1_STATUS
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."	
Register Address: 406H, 1030	IA32_MC1_ADDR
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.	
Register Address: 408H, 1032	IA32_MC2_CTL
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."	
Register Address: 409H, 1033	IA32_MC2_STATUS
See Chapter 17.3.2.2, "IA32_MCi_STATUS MSRS."	
Register Address: 40AH, 1034	IA32_MC2_ADDR
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.	
Register Address: 40CH, 1036	MSR_MC4_CTL
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."	
Register Address: 40DH, 1037	MSR_MC4_STATUS
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."	
Register Address: 40EH, 1038	MSR_MC4_ADDR

Table 2-67. MSRs in Pentium M Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.	
Register Address: 410H, 1040	MSR_MC3_CTL
See Section 17.3.2.1, "IA32_MC**i**_CTL MSRs."	
Register Address: 411H, 1041	MSR_MC3_STATUS
See Section 17.3.2.2, "IA32_MCi_STATUS MSRS."	
Register Address: 412H, 1042	MSR_MC3_ADDR
See Section 17.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.	
Register Address: 600H, 1536	IA32_DS_AREA
DS Save Area (R/W) See Table 2-2. Points to the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 21.6.3.4, "Debug Store (DS) Mechanism."	
31:0	DS Buffer Management Area Linear address of the first byte of the DS buffer management area.
63:32	Reserved.

2.22 MSRS IN THE P6 FAMILY PROCESSORS

The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as "architectural" and have had their names changed. See Table 2-2 for a list of the architectural MSRs.

Table 2-68. MSRs in the P6 Family Processors

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Register Address: 0H, 0	P5_MC_ADDR
See Section 2.23, "MSRs in Pentium Processors."	
Register Address: 1H, 1	P5_MC_TYPE
See Section 2.23, "MSRs in Pentium Processors."	
Register Address: 10H, 16	TSC
See Section 19.17, "Time-Stamp Counter."	
Register Address: 17H, 23	IA32_PLATFORM_ID
Platform ID (R) The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.	
49:0	Reserved.

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
52:50	Platform Id (R) Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
56:53	L2 Cache Latency Read.
59:57	Reserved.
60	Clock Frequency Ratio Read.
63:61	Reserved.
Register Address: 1BH, 27	APIC_BASE
Section 12.4.4, “Local APIC Status and Location.”	
7:0	Reserved.
8	Boot Strap Processor Indicator Bit 1 = BSP
10:9	Reserved.
11	APIC Global Enable Bit - Permanent till reset 1 = Enabled. 0 = Disabled.
31:12	APIC Base Address.
63:32	Reserved.
Register Address: 2AH, 42	EBL_CR_POWERON
Processor Hard Power-On Configuration (R/W) Enables and disables processor features, and (R) indicates current processor configuration.	
0	Reserved[1]
1	Data Error Checking Enable (R/W) 1 = Enabled. 0 = Disabled.
2	Response Error Checking Enable FRCERR Observation Enable (R/W) 1 = Enabled. 0 = Disabled.
3	AERR# Drive Enable (R/W) 1 = Enabled. 0 = Disabled.
4	BERR# Enable for Initiator Bus Requests (R/W) 1 = Enabled. 0 = Disabled.

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	Bit Description
5	Reserved.
6	BERR# Driver Enable for Initiator Internal Errors (R/W) 1 = Enabled. 0 = Disabled.
7	BINIT# Driver Enable (R/W) 1 = Enabled. 0 = Disabled.
8	Output Tri-state Enabled (R) 1 = Enabled. 0 = Disabled.
9	Execute BIST (R) 1 = Enabled. 0 = Disabled.
10	AERR# Observation Enabled (R) 1 = Enabled. 0 = Disabled.
11	Reserved.
12	BINIT# Observation Enabled (R) 1 = Enabled. 0 = Disabled.
13	In Order Queue Depth (R) 1 = 1. 0 = 8.
14	1-MByte Power on Reset Vector (R) 1 = 1MByte. 0 = 4GBytes.
15	FRC Mode Enable (R) 1 = Enabled. 0 = Disabled.
17:16	APIC Cluster ID (R)
19:18	System Bus Frequency (R) 00 = 66MHz. 10 = 100Mhz. 01 = 133MHz. 11 = Reserved.
21: 20	Symmetric Arbitration ID (R)
25:22	Clock Frequency Ratio (R)
26	Low Power Mode Enable (R/W)
27	Clock Frequency Ratio
63:28	Reserved.[1]
Register Address: 33H, 51	MSR_TEST_CTRL

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Test Control Register	
29:0	Reserved.
30	Streaming Buffer Disable
31	Disable LOCK# Assertion for split locked access.
Register Address: 79H, 121	BIOS_UPDT_TRIG
BIOS Update Trigger Register.	
Register Address: 88H, 136	BBL_CR_D0[63:0]
Chunk 0 data register D[63:0]: used to write to and read from the L2.	
Register Address: 89H, 137	BBL_CR_D1
Chunk 1 data register D[63:0]: used to write to and read from the L2.	
Register Address: 8AH, 138	BBL_CR_D2
Chunk 2 data register D[63:0]: used to write to and read from the L2.	
Register Address: 8BH, 139	BIOS_SIGN/BBL_CR_D3
BIOS Update Signature Register or Chunk 3 data register D[63:0]. Used to write to and read from the L2 depending on the usage model.	
Register Address: C1H, 193	PerfCtr0 (PERFCTR0)
Performance Counter Register See Table 2-2.	
Register Address: C2H, 194	PerfCtr1 (PERFCTR1)
Performance Counter Register See Table 2-2.	
Register Address: FEH, 254	MTRRcap
Memory Type Range Registers	
Register Address: 116H, 278	BBL_CR_ADDR
Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses.	
2:0	Reserved; set to 0.
31:3	Address bits [35:3].
63:32	Reserved.
Register Address: 118H, 280	BBL_CR_DECC
Data ECC register D[7:0]: used to write ECC and read ECC to/from L2.	
Register Address: 119H, 281	BBL_CR_CTL
Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response.	

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
4:0	L2 Command: 01100 = Data Read w/ LRU update (RLU). 01110 = Tag Read w/ Data Read (TRR). 01111 = Tag Inquire (TI). 00010 = L2 Control Register Read (CR). 00011 = L2 Control Register Write (CW). 010 + MESI encode = Tag Write w/ Data Read (TWR). 111 + MESI encode = Tag Write w/ Data Write (TWW). 100 + MESI encode = Tag Write (TW).
6:5	
7	State to L2
9:8	Reserved.
11:10	Way 0 - 00, Way 1 - 01, Way 2 - 10, Way 3 - 11 Way to L2
13:12	Modified - 11,Exclusive - 10, Shared - 01, Invalid - 00 Way from L2
15:14	State from L2.
16	Reserved.
17	L2 Hit.
18	Reserved.
20:19	User supplied ECC.

21	Processor number: [2]
	Disable = 1.
	Enable = 0.
	Reserved.

63:22	Reserved.
Register Address: 11AH, 282	BBL_CR_TRIG
Trigger register: used to initiate a cache configuration accesses access, Write only with Data = 0.	
Register Address: 11BH, 283	BBL_CR_BUSY
Busy register: indicates when a cache configuration accesses L2 command is in progress. D[0] = 1 = BUSY.	
Register Address: 11EH, 286	BBL_CR_CTL3
Control register 3: used to configure the L2 Cache.	
0	L2 Configured (read/write).
4:1	L2 Cache Latency (read/write).
5	ECC Check Enable (read/write).
6	Address Parity Check Enable (read/write).
7	CRTN Parity Check Enable (read/write).
8	L2 Enabled (read/write).

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
10:9	L2 Associativity (read only): 00 = Direct Mapped. 01 = 2 Way. 10 = 4 Way. 11 = Reserved.
12:11	Number of L2 banks (read only).
17:13	Cache size per bank (read/write): 00001 = 256 KBytes. 00010 = 512 KBytes. 00100 = 1 MByte. 01000 = 2 MBytes. 10000 = 4 MBytes.
18	Cache State error checking enable (read/write).
19	Reserved.
22:20	L2 Physical Address Range support: 111 = 64 GBytes. 110 = 32 GBytes. 101 = 16 GBytes. 100 = 8 GBytes. 011 = 4 GBytes. 010 = 2 GBytes. 001 = 1 GByte. 000 = 512 MBytes.
23	L2 Hardware Disable (read only).
24	Reserved.
25	Cache bus fraction (read only).
63:26	Reserved.
Register Address: 174H, 372	SYSENTER_CS_MSR
CS register target for CPL 0 code	
Register Address: 175H, 373	SYSENTER_ESP_MSR
Stack pointer for CPL 0 stack	
Register Address: 176H, 374	SYSENTER_EIP_MSR
CPL 0 code entry point	
Register Address: 179H, 377	MCG_CAP
Machine Check Global Control Register	
Register Address: 17AH, 378	MCG_STATUS
Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.	
Register Address: 17BH, 379	MCG_CTL
Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).	
Register Address: 186H, 390	PerfEvtSel0 (EVNTSEL0)

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Performance Event Select Register 0 (R/W)	
7:0	Event Select Refer to Performance Counter section for a list of event encodings.
15:8	UMASK (Unit Mask) Unit mask register set to 0 to enable all count options.
16	USER Controls the counting of events at Privilege levels of 1, 2, and 3.
17	OS Controls the counting of events at Privilege level of 0.
18	E Occurrence/Duration Mode Select: 1 = Occurrence. 0 = Duration.
19	PC Enabled the signaling of performance counter overflow via BP0 pin.
20	INT Enables the signaling of counter overflow via input to APIC: 1 = Enable. 0 = Disable.
22	ENABLE Enables the counting of performance events in both counters: 1 = Enable. 0 = Disable.
23	INV Inverts the result of the CMASK condition: 1 = Inverted. 0 = Non-Inverted.
31:24	CMASK (Counter Mask)
Register Address: 187H, 391	PerfEvtSel1 (EVNTSEL1)
Performance Event Select for Counter 1 (R/W)	
7:0	Event Select Refer to Performance Counter section for a list of event encodings.
15:8	UMASK (Unit Mask) Unit mask register set to 0 to enable all count options.
16	USER Controls the counting of events at Privilege levels of 1, 2, and 3.
17	OS Controls the counting of events at Privilege level of 0.

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
18	E Occurrence/Duration Mode Select: 1 = Occurrence. 0 = Duration.
19	PC Enabled the signaling of performance counter overflow via BP0 pin.
20	INT Enables the signaling of counter overflow via input to APIC. 1 = Enable. 0 = Disable.
23	INV Inverts the result of the CMASK condition. 1 = Inverted. 0 = Non-Inverted.
31:24	CMASK (Counter Mask)
Register Address: 1D9H, 473	DEBUGCTLMSR
Enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode.	
0	Enable/Disable Last Branch Records
1	Branch Trap Flag
2	Performance Monitoring/Break Point Pins
3	Performance Monitoring/Break Point Pins
4	Performance Monitoring/Break Point Pins
5	Performance Monitoring/Break Point Pins
6	Enable/Disable Execution Trace Messages
31:7	Reserved.
Register Address: 1DBH, 475	LASTBRANCHFROMIP
32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.	
Register Address: 1DCH, 476	LASTBRANCHTOIP
32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.	
Register Address: 1DDH, 477	LASTINTFROMIP
Last INT from IP	
Register Address: 1DEH, 478	LASTINTTOIP
Last INT to IP	
Register Address: 200H, 512	MTRRphysBase0
Memory Type Range Registers	
Register Address: 201H, 513	MTRRphysMask0
Memory Type Range Registers	

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Register Address: 202H, 514	MTRRphysBase1
Memory Type Range Registers	
Register Address: 203H, 515	MTRRphysMask1
Memory Type Range Registers	
Register Address: 204H, 516	MTRRphysBase2
Memory Type Range Registers	
Register Address: 205H, 517	MTRRphysMask2
Memory Type Range Registers	
Register Address: 206H, 518	MTRRphysBase3
Memory Type Range Registers	
Register Address: 207H, 519	MTRRphysMask3
Memory Type Range Registers	
Register Address: 208H, 520	MTRRphysBase4
Memory Type Range Registers	
Register Address: 209H, 521	MTRRphysMask4
Memory Type Range Registers	
Register Address: 20AH, 522	MTRRphysBase5
Memory Type Range Registers	
Register Address: 20BH, 523	MTRRphysMask5
Memory Type Range Registers	
Register Address: 20CH, 524	MTRRphysBase6
Memory Type Range Registers	
Register Address: 20DH, 525	MTRRphysMask6
Memory Type Range Registers	
Register Address: 20EH, 526	MTRRphysBase7
Memory Type Range Registers	
Register Address: 20FH, 527	MTRRphysMask7
Memory Type Range Registers	
Register Address: 250H, 592	MTRRfix64K_00000
Memory Type Range Registers	
Register Address: 258H, 600	MTRRfix16K_80000
Memory Type Range Registers	
Register Address: 259H, 601	MTRRfix16K_A0000
Memory Type Range Registers	
Register Address: 268H, 616	MTRRfix4K_C0000
Memory Type Range Registers	
Register Address: 269H, 617	MTRRfix4K_C8000

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Memory Type Range Registers	
Register Address: 26AH, 618	MTRRfix4K_D0000
Memory Type Range Registers	
Register Address: 26BH, 619	MTRRfix4K_D8000
Memory Type Range Registers	
Register Address: 26CH, 620	MTRRfix4K_E0000
Memory Type Range Registers	
Register Address: 26DH, 621	MTRRfix4K_E8000
Memory Type Range Registers	
Register Address: 26EH, 622	MTRRfix4K_F0000
Memory Type Range Registers	
Register Address: 26FH, 623	MTRRfix4K_F8000
Memory Type Range Registers	
Register Address: 2FFH, 767	MTRRdefType
Memory Type Range Registers	
2:0	Default memory type
10	Fixed MTRR enable
11	MTRR Enable
Register Address: 400H, 1024	MC0_CTL
Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).	
Register Address: 401H, 1025	MC0_STATUS
Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.	
15:0	MC_STATUS_MCACOD
31:16	MC_STATUS_MSCOD
57	MC_STATUS_DAM
58	MC_STATUS_ADDRV
59	MC_STATUS_MISCV
60	MC_STATUS_EN. (Note: For MC0_STATUS only, this bit is hardcoded to 1.)
61	MC_STATUS_UC
62	MC_STATUS_O
63	MC_STATUS_V
Register Address: 402H, 1026	MC0_ADDR
Register Address: 403H, 1027	MC0_MISC
Defined in MCA architecture but not implemented in the P6 family processors.	
Register Address: 404H, 1028	MC1_CTL

Table 2-68. MSRs in the P6 Family Processors (Contd.)

Register Address: Hex, Decimal	Register Name
Register Information / Bit Fields	**Bit Description**
Register Address: 405H, 1029	MC1_STATUS
Bit definitions same as MC0_STATUS.	
Register Address: 406H, 1030	MC1_ADDR
Register Address: 407H, 1031	MC1_MISC
Defined in MCA architecture but not implemented in the P6 family processors.	
Register Address: 408H, 1032	MC2_CTL
Register Address: 409H, 1033	MC2_STATUS
Bit definitions same as MC0_STATUS.	
Register Address: 40AH, 1034	MC2_ADDR
Register Address: 40BH, 1035	MC2_MISC
Defined in MCA architecture but not implemented in the P6 family processors.	
Register Address: 40CH, 1036	MC4_CTL
Register Address: 40DH, 1037	MC4_STATUS
Bit definitions same as MC0_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.	
Register Address: 40EH, 1038	MC4_ADDR
Defined in MCA architecture but not implemented in P6 Family processors.	
Register Address: 40FH, 1039	MC4_MISC
Defined in MCA architecture but not implemented in the P6 family processors.	
Register Address: 410H, 1040	MC3_CTL
Register Address: 411H, 1041	MC3_STATUS
Bit definitions same as MC0_STATUS.	
Register Address: 412H, 1042	MC3_ADDR
Register Address: 413H, 1043	MC3_MISC
Defined in MCA architecture but not implemented in the P6 family processors.	

NOTES

1. Bit 0 of this register has been redefined several times, and is no longer used in P6 family processors.
2. The processor number feature may be disabled by setting bit 21 of the BBL_CR_CTL MSR (model-specific register address 119h) to "1". Once set, bit 21 of the BBL_CR_CTL may not be cleared. This bit is write-once. The processor number feature will be disabled until the processor is reset.
3. The Pentium III processor will prevent FSB frequency overclocking with a new shutdown mechanism. If the FSB frequency selected is greater than the internal FSB frequency the processor will shutdown. If the FSB selected is less than the internal FSB frequency the BIOS may choose to use bit 11 to implement its own shutdown policy.

2.23 MSRS IN PENTIUM PROCESSORS

The following MSRs are defined for the Pentium processors. The P5_MC_ADDR, P5_MC_TYPE, and TSC MSRs (named IA32_P5_MC_ADDR, IA32_P5_MC_TYPE, and IA32_TIME_STAMP_COUNTER in the Pentium 4 processor) are architectural; that is, code that accesses these registers will run on Pentium 4 and P6 family processors without generating exceptions (see Section 2.1, "Architectural MSRs"). The CESR, CTR0, and CTR1 MSRs are unique to Pentium processors; code that accesses these registers will generate exceptions on Pentium 4 and P6 family processors.

Table 2-69. MSRs in the Pentium Processor

Register Address: Hex, Decimal	Register Name
Register Information	
Register Address: 0H, 0	P5_MC_ADDR
See Section 17.10.2, “Pentium Processor Machine-Check Exception Handling.”	
Register Address: 1H, 1	P5_MC_TYPE
See Section 17.10.2, “Pentium Processor Machine-Check Exception Handling.”	
Register Address: 10H, 16	TSC
See Section 19.17, “Time-Stamp Counter.”	
Register Address: 11H, 17	CESR
See Section 21.6.9.1, “Control and Event Select Register (CESR).”	
Register Address: 12H, 18	CTR0
Section 21.6.9.3, “Events Counted.”	
Register Address: 13H, 19	CTR1
Section 21.6.9.3, “Events Counted.”	