Intel® Quark™ microcontroller D1000

Programmer's Reference Manual

November 2015

Document Number: 332913-002US

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2015	002	Revised table 11 IOAPIC Memory Mapped Registers
October 2015	001	Initial release

§ §

1.0 Introduction

This document describes the external architecture of the Intel® Quark™ microcontroller D1000 processor. This description includes core operation, external interfaces, register definitions, etc. This document is intended as a reference for a logic design group, architecture validation, firmware development, software device developers, test engineers or anyone who may need specific technical or programming information about the Intel® Quark™ microcontroller D1000.

1.1 Intel® Quark™ microcontroller D1000 CPU Overview

Important characteristics of the Intel® Quark™ microcontroller D1000 CPU are provided in the following list:

- 32-bit processor core
- IA-32 instruction encoding
- 5 stage pipeline
- Harvard architecture
- 8KB of on-chip data SRAM
- 32KB of on-chip data/execution FLASH
- Deterministic 21 Cycle interrupt latency
- Minimal processor initialization for fast power-up

1.2 Interrupts

The CPU implements an Advanced Programmable Interrupt Controller (APIC) with an integrated IOAPIC. The CPU routes incoming interrupts via an Interrupt Descriptor Table (IDT). The IOAPIC is tightly coupled with the local APIC. The IOAPIC supports external interrupts that map to the Interrupt Descriptor Table (IDT) starting at vector 20h. Vectors 0 to 1Fh are reserved for processor exceptions.

1.3 I/O

All I/O interaction occurs via Memory Mapped I/O (MMIO). MMIO device registers map into the Strongly Ordered memory range as described in "Memory Ordering" on page 16.

1.4 Code and Data Interfaces

The CPU uses a Harvard architecture, which means separate physical interfaces for code and data. Data interfaces are 32-bits wide, support read-modify-write transactions efficiently and allow memory modification at byte granularity. The instruction interface provides a 16 byte fetch width. Due to the variable length instruction set of the CPU, a wider instruction fetch path improves performance. This issue is of particular importance for branch performance in which the pipeline must restart instruction fetch at the branch target address.

1.4.1 Instruction Alignment

The CPU imposes no instruction alignment restrictions. However, alignment can affect hardware instruction fetch efficiency, particularly alignment of the target of jump or call instructions. For these cases, instruction alignment up to an 8 byte boundary may improve efficiency.

Note: RTL simulators often assert on a read from uninitialized memory. This may occur when an instruction fetch near the end of the elf code segment reads uninitialized memory following the last instruction byte. Pad the code segment using linker script commands to avoid this problem.

1.4.2 Data Alignment

The CPU imposes no data alignment restrictions. When fetching arbitrary data, the CPU performs one or possibly two reads from 4 byte aligned addresses. To maximize efficiency, software should arrange data items on natural boundaries up to a maximum alignment of 4 bytes.

1.4.3 Stack Alignment

As with data accesses, the Intel® Quark™ microcontroller D1000 CPU does not impose alignment restrictions on the stack pointer (ESP). However, a stack pointer that is not aligned with respect to push/pop size imposes a significant efficiency penalty. Software should maintain the stack on 4 byte boundary.

1.5 Floating Point

The CPU does not implement hardware floating point support. The compiler provides a software implementation of floating point functions transparently to the C/C++ programmer.

§ §

2.0 Compatibility

The CPU borrows IA-32 instruction encoding, but is not an IA-32 processor and is not compatible with existing IA-32 applications or operating systems. Specifically, the Intel® Quark™ microcontroller D1000 CPU supports only a subset of the full IA-32 instruction set. Likewise, the CPU architecture excludes many legacy features such as segmentation. The CPU implements system software features not available or solved differently on IA-32. Software written for IA-32 processors requires porting to the Intel® Quark™ microcontroller D1000.

§ §

3.0 Memory Model

The CPU provides a simple linear physical 32-bit memory model. The CPU does not support any form of memory address segmentation. The following sections provide additional detail.

3.1 Bit and Byte Order

The CPU uses *little-endian* byte order. See Figure 1.

Figure 1. CPU Byte Order that Follows the Little-Endian Convention

31 … 24	23 … 16	15 … 8	7 … 0	
Byte 15 highest address	Byte 14	Byte 13	Byte 12	0Ch
Long 0				08h
Word 1		Word 0		04h
Byte 3	Byte 2	Byte 1	Byte 0 lowest address	00h

The CPU supports 8-bit (byte), 16-bit (word) and 32-bit (dword) data accesses. The CPU does not support 64-bit (qword) access. Instructions performing 16-bit data accesses require a 66h instruction prefix byte. In general, the 66h prefix provides an operand size override for most data or register access instructions.

3.2 Addressing

The CPU uses flat and physical addressing for memory. Flat means that the CPU does not use any form of memory segmentation. Physical means the CPU does not perform memory address translations. Software uses physical memory addresses.

3.3 Memory Ordering

The CPU supports two memory ordering models, Strongly Ordered and Weakly ordered. The CPU differentiates between Weakly Ordered and Strongly Ordered memory by the highest address bit. Thus Memory-Mapped IO devices appear at addresses higher than 80000000h as shown in Table 1.

Memory located in the Weakly Ordered memory range must be free of side effects. Thus, a read or write to an address in Processor Ordered memory must not affect the contents of a different address in Processor Ordered memory or any other memory region. This guarantee allows the processor to more efficiently access Processor Ordered memory. The CPU may perform speculative reads in Processor Ordered memory.

A read or write to Strongly Ordered memory need not be free of side-effects. Thus, a read or write to an address in Strongly Ordered memory may affect the content of a different Strongly Ordered memory address. A read or write to Strongly Ordered memory must not affect the content of Processor Ordered memory.

Table 1. Strong and Weak Order Memory

Address Range	Memory Ordering Model
FFFFFFFFh . . . 80000000h	Strongly Ordered
7FFFFFFFh . . . 00000000h	Weakly Ordered

3.3.1 Strong Ordering Rules

For Strongly Ordered accesses, the CPU issues reads and writes on the external memory interface in the same order encountered in the instruction stream.

3.3.2 Weak Ordering Rules

For accesses to Weakly Ordered memory, the following rules apply.

- Reads are not reordered with other reads.
- Writes are not reordered with other writes
- Writes are not reordered with older reads.
- Reads may be reordered with older writes to different locations but not with older writes to the same location
- Reads or writes cannot be reordered with respect to serializing instructions.

3.3.3 Mixed Ordering Rules

For access sequences involving both Weakly Ordered memory and Strongly Ordered memory, the following rules apply.

- Writes to Weakly Ordered memory are not reordered with respect to Strongly Ordered writes.
- Reads to Weakly Ordered memory may be reordered with respect to Strongly Ordered reads or writes.

3.3.4 Write Flushing

Writes to MMIO registers in devices may traverse a variety of intermediate buffers depending on the nature of the embedded design. These buffers may not be visible to the CPU. If software requires a strongly ordered write to take immediate effect, then software must cause a write flush. The recommended method is to follow a strongly ordered write with a read to the same MMIO address.

3.4 Self-Modifying Code

Except for bulk FLASH reprogramming, the CPU cannot create self-modifying code. The CPU cannot execute out of on-chip SRAM.

3.5 Stack Behavior

The CPU uses a grow-down stack. The CPU follows decrement-then-write behavior for pushes and read-then-increment behavior for pops. The CPU stack pointer register is ESP. Other than being the implied pointer in stack specific instructions, the %esp register behaves as a general purpose register.

3.5.1 Stack Alignment

As with data accesses, the CPU does not impose alignment restrictions on the stack pointer (ESP). However, a stack pointer that is not aligned with respect to push/pop size imposes an efficiency penalty. Software should maintain the stack on 4 byte boundary.

Note that the PUSH instructions are irregular with regard to stack alignment. 8-bit push instructions sign extend the value to enforce stack alignment but 16-bit push instructions do not sign extend and cause an unaligned stack. See Section 8.49 for more information.

3.5.2 Stack Over/Underflow

In general, stack over/underflow behaves like an errant data pointer bug.

§ §

4.0 Registers

The CPU defines 7 general purpose registers, a stack pointer and an instruction pointer. The CPU also implements several other system support registers such as a supervisor stack pointer.

4.1 General Purpose Registers

The CPU 32-bit general purpose registers (see Figure 2) have 8-bit and 16-bit renames as shown. The 16-bit forms of EAX, EBX, ECX and EDX are AX, BX, CD, DX respectively.

Note: 16-bit wide accesses requires the 66h prefix on the instruction. 32-bit and 8-bit forms are encoded without a prefix.

Figure 2. General Purpose Registers

	31 … 16	15 … 8	7 … 0
EAX		AH	AL
EBX		BH	BL
ECX		CH	CL
EDX		DH	DL
ESI		SI	
EDI		DI	
EBP		BP	
ESP	Stack Pointer		

The instruction opcode specifies the effective width of the register as either an 8-bit or 32-bit form. The 66h prefix provides an operand width override which converts the 32-bit operand form into a 16-bit operand form.

4.2 Special Purpose Registers

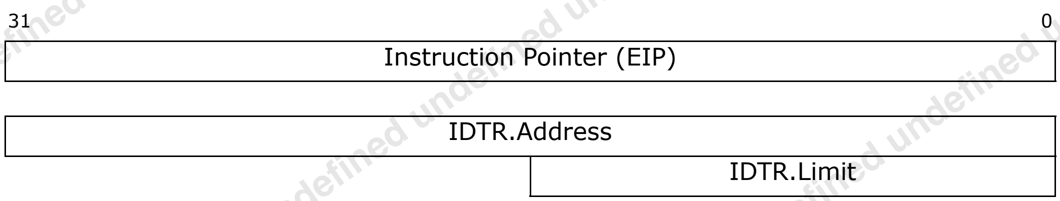
In addition to the general purpose registers, the CPU defines several special purpose registers. See Figure 3.

The IDTR Address register contains the starting address of the Interrupt Descriptor Table (IDT). The IDTR Limit register contains the size in bytes of the IDT. The IDTR Limit register allows software to reduce the memory footprint of the IDT by eliminating unneeded vectors. For more information, see Section 8.38 which describes initialization of this register.

If an external interrupt or INT instruction requires a vector beyond the byte limit in the IDTR Limit register, the CPU generates a General Protection Fault (#GP) with the IDT flag set in the error code. See Section 5.12. The exception handling algorithm in Figure 15, Figure 16 and Figure 17 provide additional detail.

The Interrupt Descriptor Table Register (IDTR) is split into a 32-bit Address field and a 16-bit Limit field.

Figure 3. Special Purpose Registers



4.3 EFLAGS

The CPU supports a status register called EFLAGS as shown in Figure 4 and Table 2.

The CPU reserves EFLAGS bits shaded gray. For a comparison with IA-32, refer to Appendix A.8. Status flags represent the status of arithmetic operations or other cases that can be manipulated by user-mode processes. Fixed flags are read-only and do not change state. System flags r present processor state that cannot be altered by a user-mode process. Writes in user-mode to these bits are ignored. Reserved flags cannot be altered by a user or supervisor mode process. Writes to these bits generate a General Protection Fault (#GP).

Figure 4. Flags Defined in the EFLAGS Register

Flag	Bit	Type	Description
CF	0	Status	Carry Flag
	1	Fixed	Always 1
	2	Reserved	
	5-3	Reserved	
ZF	6	Status	Zero Flag

Flag	Bit	Type	Description
SF	7	Status	Sign Flag
TF	8	System	Trap Flag
IF	9	System	Interrupt Enable Flag
	10	Reserved	
OF	11	Status	Overflow Flag
	12-31	Reserved	

Table 2. FLAG Detailed Descriptions

Flag	Description
CF	Carry Flag - The CPU sets this flag if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; The CPU clears CF otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. CF is also used in multiple precision arithmetic. Software may manipulate the CF directly using the STC, CLC, and CMC instructions.
ZF	Zero Flag - The CPU sets this flag if the result of the operation is zero; The CPU clears ZF otherwise.
SF	Sign Flag - The CPU sets this flag equal to the most-significant bit of the result, which is the sign bit of a signed integer. A 0 indicates a positive value and 1 indicates a negative value.
OF	Overflow Flag - The CPU sets this flag if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand. The CPU clears OF otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.
TF	Trap Flag - Software sets this flag to enable single-step mode for debugging. Software clears TF to disable single-step mode. In single step mode, the CPU generates a debug exception after each instruction. This allows the execution state of a program to be inspected after each instruction. If software sets the TF flag using a POPFD or IRET instruction, the CPU generates a debug exception after the instruction that follows the POPFD or IRET. When accessing an exception or interrupt handler through either an interrupt gate or a trap gate, the CPU clears the TF flag in the EFLAGS register after saving the contents of the EFLAGS register on the stack. Clearing the TF flag prevents instruction tracing from affecting interrupt response. A subsequent IRET instruction restores TF to the value in the saved contents of the EFLAGS register on the stack.
IF	Interrupt Enable Flag - This flag controls the response of the processor to maskable hardware interrupt requests. Software sets IF using the STI instruction to respond to maskable hardware interrupts. Software clears the IF flag with the CLI instruction to inhibit maskable hardware interrupts. Similarly, the IRET and POPFD instructions load EFLAGS from the stack, including the IF flag value. The CPU clears the IF flag on an interrupt through an interrupt gate.

§ §

5.0 Exceptions

An exception is a discontinuity in the instruction stream to handle unusual circumstances or external events. The CPU implements an exception handling architecture based on an Exception Processing Unit (EPU), an Advanced Programmable Interrupt Controller (APIC) and integrated IOAPIC. The EPU directs exception handling by means of a memory resident Interrupt Descriptor Table (IDT) which is controlled by software. The APIC and IOAPIC provide an interface to external interrupt sources as described in Chapter 7.0, "APIC and IOAPIC" on page 44. Because the CPU eliminates segmentation and other overheads, interrupt processing requires approximately 21 cycles from assertion of an interrupt at the IOAPIC input to execution of the first instruction of the interrupt handler.

5.1 Exception Types

The CPU supports interrupts, faults, traps and aborts. The CPU treats faults and traps as synchronous exceptions associated with a specific instruction. Interrupts and aborts are not associated with a specific instruction.

When an exception occurs, the CPU's Exception Processing Unit (EPU) redirects execution to the appropriate exception handler routine. System software specifies exception handler entry points via a Interrupt Descriptor Table (IDT) in memory. Software executing in supervisor mode loads the location of the IDT using the LIDT instruction.

5.1.1 Interrupts

An interrupt is an external asynchronous event routed to the CPU through the APIC, e.g. device and timer interrupts.

5.1.2 Faults

A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (EIP in the stack frame) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.

For a Not-Present Fault (#NP) or General Protection Fault (#GP), the CPU pushes an additional 32-bit error code in the exception stack frame. The error code allows software to resolve ambiguities regarding the source of the #NP or #GP.

For a Machine Check Fault (#MC), the CPU supports an additional 32-bit error code and a 32-bit address on the exception stack frame.

5.1.3 Traps

A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler (EIP in the stack frame) points to the instruction to be executed after the trapping instruction.

If the CPU detects a trap for an instruction which transfers execution, the return instruction pointer (EIP in the stack frame) reflects the transfer. For example, if a trap is detected while executing a JMP instruction, the return instruction pointer points to the destination of the JMP instruction, not to the next address past the JMP instruction.

5.1.4 Aborts

An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. The CPU uses aborts to report severe errors, such as double faults.

5.2 Exception Handling

After recognizing an exception, the CPU saves context information to the stack, then jumps to the address specified by the matching IDT entry. The format of the saved stack frame depends on the nature of the exception. The sections describing each exception provide specific stack frame information.

5.3 Triple Fault

The CPU generates a Triple Fault when unable to process a Double Fault (#DF) due to problems in the Interrupt Descriptor Table (IDT). On a Triple Fault, the CPU takes the following actions:

- Enters the stopped state
- Asserts the CPU_ERR output signal

Exit from the stopped state is by an external hardware signal only, specifically, one of the following.

- Power cycle
- External reset
- Reset from the Debug Controller
- Reset from the Watchdog Timer

In the stopped state, the CPU does not respond to external interrupts. The CPU clears the CPU_ERR output only on reset. Chapter 6.0 describes the reset process.

Triple Fault conditions often occur during early software development in which the developer has not yet implemented exception handling. In such cases, any exception becomes a Triple Fault due to an absent or uninitialized IDT.

5.4 Interrupt Descriptor Table

Software specifies all interrupt handlers in the Interrupt Descriptor Table (IDT). During exception processing, the Exception Processing Unit (EPU) reads the IDT Entry associated with the pending exception. During initialization, software loads the Interrupt Descriptor Table Register (IDTR) structure described in Section 8.38, "LIDT - Load Interrupt Descriptor Table Register" on page 82. The IDTR specifies the base physical address and the number of entries in the IDT. Table 3 shows the layout of the IDT. By convention, vectors 0 to 31 are reserved for processor exceptions.

Table 3. Interrupt Descriptor Table (IDT)

Vector	Name	Type	Error Code?	Description
0	#DE	Fault	No	Divide by 0
1	#DB	Trap	No	Debug Exception
2		Reserved		
3	#BP	Trap	No	Breakoutpoint(INT3)
4 - 5		Reserved		
6	#UD	Fault	No	Invalid Opcode
7		Reserved		
8	#DF	Abort	Yes	Double Fault
9 - 10		Reserved		
11	#NP	Fault	Yes	Not Present
12		Reserved		
13	#GP	Fault	Yes	General Protection
14 - 17		Reserved		
18	#MC	Abort	Yes	Machine Check
19 - 31		Reserved		
32 - 255		Interrupt	No	Asynchronous IRQ

Note: Each entry in Table 3 occupies 8 bytes. For a comparison with IA-32 exception vectors, refer to Section A.9, "Exceptions" on page 107.

5.5 Format of Interrupt Descriptors

Figure 5 shows the format of the CPU interrupt descriptors. These structures differ only in bit 8 which differentiates traps from interrupts. The CPU generates a General Protection Fault (#GP) when the requested vector lies outside the range of the Interrupt Descriptor Table.

Figure 5. CPU Interrupt and Trap Descriptor Format

31 – 16	15	14	13	12	11	10	9	8	7 – 0	
Address 31-16	P	0	0	0	1	1	1	0	0	04h
0	Address 15-0									00h

Note: Shaded areas are reserved and software must set these bits as shown in Figure 5.

Table 4. CPU Interrupt and Trap Descriptions

Field	Description
Address	Software sets this field to the EIP of the interrupt service routine for this vector. The descriptor splits this field into high and low halves.
P	Present - Software sets this bit to 1 for valid descriptors that contain vector and 0 for invalid descriptors that do not contain a vector. The IDTR described with the LIDT instruction specifies the total number of descriptors, up to the maximum of 256. Vectors greater than the IDTR limit are automatically invalid. The CPU generates a General Protection Fault (#GP) for exceptions to a vector with an invalid descriptor.

5.6 Exception 0 - Divide Error (#DE)

The #DE fault indicates the divisor operand for a DIV or IDIV instruction is 0 or that the result cannot be represented in the number of bits specified for the destination operand.

Figure 6. Exception Frame Saved on the Stack for the #DE Exception

31 … 0	
EFLAGS	ESP+8
0/Ignored	ESP+4
EIP	ESP

5.6.1 Exception Class

Fault.

5.6.2 Error Code

None.

5.6.3 Saved Instruction Pointer

The exception stack frame contains the EIP of the instruction that generated the exception.

5.6.4 Program State Change

A program-state change does not accompany this exception, because the exception occurs before the CPU executes the faulting instruction.

5.7 Exception 1 - Debug Exception (#DB)

The CPU generates a #DB trap after retirement of every instruction while executing in Software Single-Step (SWSS) mode. Software enables SWSS mode by setting the Trap Flag (EFLAGS.TF).

Figure 7. Exception Frame Saved on the Stack for the #DB Exception

31 ... 0	
EFLAGS	ESP+8
0/Ignored	ESP+4
EIP	ESP

Note: The CPU also supports In-Circuit Emulation Single Step (ICESS) capability provided by the Debug Controller. The Debug Controller provides a hardware based mechanism to place the CPU in ICESS mode without support from software in the target platform. In this case, the CPU does not generate a #DB exception, but instead enters Probe Mode and transfers control to the Debug Controller. In Probe Mode, the CPU interacts with a debugger via a JTAG interface. For more information, refer to the Intel® Quark™ microcontroller D1000 User Guide.

5.7.1 Exception Class

Trap.

5.7.2 Error Code

None.

5.7.3 Saved Instruction Pointer

The exception stack frame contains the EIP of the instruction following the trapping instruction.

5.7.4 Program State Change

The state of the program is essentially unchanged because the #DB trap does not affect any register or memory locations. A debugger can resume the software process by executing IRET.

5.8 Exception 3 - Breakpoint (#BP)

#BP indicates that the CPU executed a breakpoint instruction (INT3), resulting in a breakpoint trap. Typically, a debugger sets a breakpoint by replacing the first opcode byte of an instruction with the opcode for the INT3 instruction. The INT3 instruction is one byte long, to simplify opcode replacement.

Software may invoke the #BP exception using either the 1 or 2 byte INT instruction forms. These are 'CC' and 'CD 03' respectively. Both instruction forms behave identically.

Note: For breakpoint support, the CPU offers debug registers accessible via the JTAG interface. Debug registers are much more convenient than injecting INT3 into the instruction stream. If more breakpoints are needed beyond what the debug registers allow, software may still rely on INT3.

5.8.1 Exception Stack Frame

Figure 8. Exception Frame Saved on the Stack for the #BP Exception

31 … 0	
EFLAGS	ESP+8
0/Ignored	ESP+4
EIP	ESP

5.8.2 Exception Class

Trap.

5.8.3 Error Code

None.

5.8.4 Saved Instruction Pointer

The exception stack frame contains the EIP of the instruction following the trapping instruction.

5.8.5 Program State Change

Even though the EIP points to the instruction following the breakpoint instruction, the state of the program is essentially unchanged because the INT3 instruction does not affect any register or memory locations. A debugger can resume the software process by replacing the INT3 instruction that caused the breakpoint with the original opcode and decrementing the EIP register value saved in the stack frame. In this case, IRET resumes program execution at the replaced instruction.

5.9 Exception 6 - Invalid Opcode (#UD)

#UD indicates that the CPU did one of the following things:

- Attempted to execute an invalid or reserved opcode.
- Attempted to execute an instruction with an operand type that is invalid for its accompanying opcode.
- Executed a UD2 instruction.
- An instruction repeats a prefix byte, such as 66 66. Refer to Section 8.2, "Instruction Prefixes" on page 52.

5.9.1 Exception Stack Frame

Figure 9. Exception Frame Saved on the Stack for the #UD Exception

31 ... 0	
EFLAGS	ESP+8
0/Ignored	ESP+4
EIP	ESP

5.9.2 Exception Class

Fault.

5.9.3 Error Code

None.

5.9.4 Saved Instruction Pointer

The exception stack frame contains the EIP of the instruction that generated the exception.

5.9.5 Program State Change

A program-state change does not accompany this exception, because the exception occurs before the CPU executes the faulting instruction.

5.10 Exception 8 - Double Fault (#DF)

#DF indicates that the CPU detected a second exception while calling an exception handler for a prior exception. Normally, when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. The CPU generates a Double Fault when the two exceptions cannot be processed serially.

See the interrupt entry algorithms in Section 5.16, "Logical Algorithms" on page 37 for the precise circumstances that generate #DF.

5.10.1 Exception Stack Frame

Figure 10. Exception Frame Saved on the Stack for the #DF Exception

31 ... 0	
EFLAGS	ESP+12
0/Ignored	ESP+8
EIP	ESP+4
0	ESP

Note: The Error Code field is always 0.

5.10.2 Exception Class

Abort.

5.10.3 Error Code

The CPU always pushes an error code of zero. Software must pop the error code from the stack before returning from the exception service routine. The stack pointer (ESP) must point to the EIP field of the stack frame before executing IRET.

5.10.4 Saved Instruction Pointer

EIP in the stack frame is undefined.

5.10.5 Program State Change

Software process state following a Double Fault is undefined. The software processes cannot be resumed or restarted. The only available action of the Double Fault exception handler is to collect all possible context information for use in diagnostics and reset the CPU.

5.11 Exception 11 - Not Present (#NP)

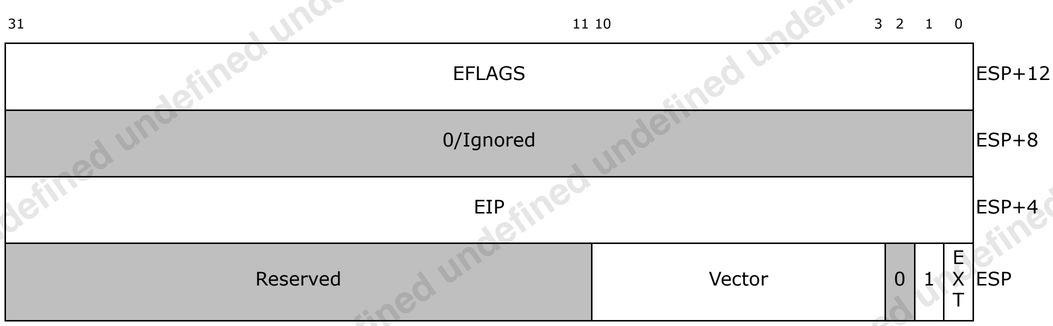
#NP indicates that an exception occurred and the corresponding Interrupt Descriptor Table Entry for that exception has the ’P’ bit clear, indicating not present.

See the interrupt entry algorithms in Section 5.16, “Logical Algorithms” on page 37 for the precise circumstances that generate #NP.

Note that if the exception vector number is larger than the size of the IDT table, then the CPU generates a General Protection Fault (#GP), and not #NP.

5.11.1 Exception Stack Frame

Figure 11. Exception Frame Saved on the Stack for the #NP Exception



Note: Software should not alter the value of the reserved field.

Table 5. Exception Stack Frame Description

Field	Description
Vector	This field contains the 8-bit index of the Interrupt Descriptor Table (IDT) Entry that caused the exception.
EXT	External Flag - The CPU sets this bit to indicate that the exception occurred during delivery of an event external to the program, e.g. an interrupt.

Note: ERRATA: For this exception, the EXT bit in the error code field is incorrect. Do not rely on this bit.

5.11.2 Exception Class

Fault.

5.11.3 Error Code

The CPU pushes an error code containing the vector number of the exception that caused the #NP.

Software must pop the error code from the stack before returning from the exception service routine. The stack pointer (ESP) must point to the EIP field of the stack frame before executing IRET.

5.11.4 Saved Instruction Pointer

If the #NP is the result of instruction execution, then EIP points to the instruction that initiated the exception. Otherwise, the #NP is the result of an external interrupt and EIP points to the next instruction the CPU will execute on return from interrupt.

5.11.5 Program State Change

A process state change does not accompany the exception. Recovery from this exception is possible by setting the present flag in the gate descriptor.

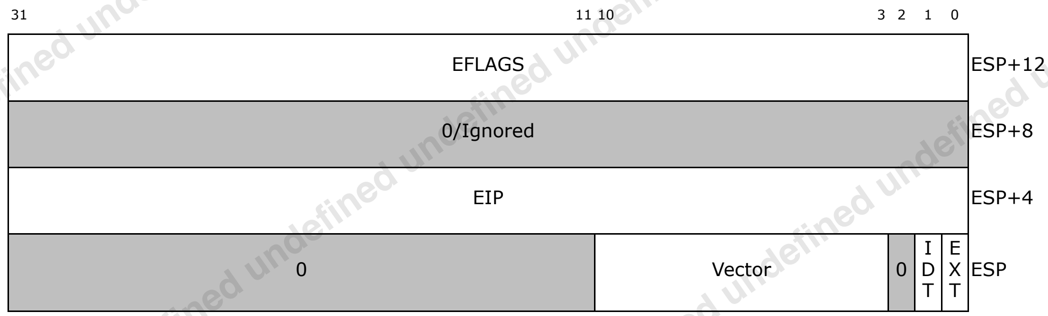
5.12 Exception 13 - General Protection (#GP)

The CPU generates #GP in the following cases:

- An exception occurred with a vector number larger than the size of the IDT table.
- An exception occurred and the corresponding IDT entry is not an Interrupt or Trap gate.
- An exception occurs during interrupt or exception entry, such as a bus error.
- Attempt to set a reserved EFLAGS bit.

5.12.1 Exception Stack Frame

Figure 12. Exception Frame Saved on the Stack for the #DF Exception



Note: The precise content of the Error Code field depends on the source of the #GP fault as described in this section. Software should not alter the value of the reserved field.

Table 6. Exception Stack Frame Description

Field	Description
Vector	This field contains the 8-bit index of the Interrupt Descriptor Table (IDT) Entry that caused the exception if the IDT Flag is 1. If the IDT Flag is 0, then this field is reserved.
IDT	IDT Flag - The CPU sets this bit to indicate the exception is associated with an error in the IDT. In this case, the Vector field is valid. The CPU clears this bit otherwise.
EXT	External Flag - The CPU sets this bit to indicate that the exception occurred during delivery of an event external to the program, e.g. an interrupt.

Note: ERRATA: For this exception, the EXT and IDT bits in the error code field are in correct. Do not rely on these bits.

5.12.2 Exception Class

Fault.

5.12.3 Error Code

The CPU pushes an error code for #GP. If the fault is associated with an IDT entry, the CPU pushes an error code containing the vector number of the exception that caused the #GP. For all other cases, the CPU pushes an error code of 0.

Software must pop the error code from the stack before returning from the exception service routine. The stack pointer (ESP) must point to the EIP field of the stack frame before executing IRET.

5.12.4 Saved Instruction Pointer

If the #GP is the result of instruction execution, then the EIP points to the instruction that initiated the exception. If the #GP is the result of an external interrupt, then the EIP points to the next instruction the CPU will execute on return from interrupt. Otherwise, the EIP points to the instruction that generated the fault.

5.12.5 Program State Change

In general, a state change does not accompany a #GP, because the CPU does not execute the invalid instruction or operation. An exception handler can be designed to correct all of the conditions that cause general-protection exceptions and resume the software process without any loss of program continuity.

5.13 Exception 18 - Machine Check (#MC)

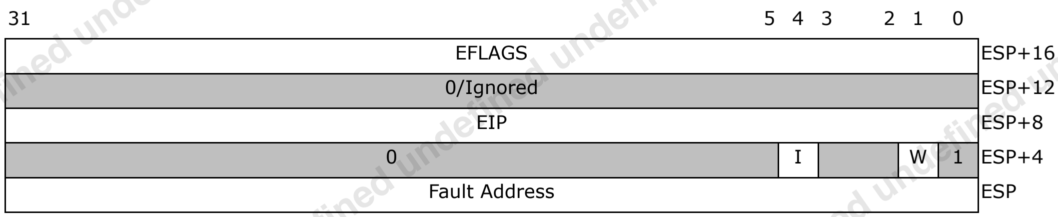
The CPU generates #MC faults in response to errors detected by hardware. Currently, the only source of the #MC fault is the CPU's BUS_ERR input on any of the CPU's memory interfaces. Hardware external to the CPU may assert the BUS_ERR input in response to an erroneous read or write transaction. The exact reason for asserting the BUS_ERR input is hardware dependent, but could for example include fundamental memory transaction errors such as writes to ROM.

While software may be able to implement system recovery in some platform specific cases, the #MC exception is an Abort class exception. In general, software does not have enough information to recover a system to a known good state after a #MC.

The two sources of #MC are in attempting to fetch an instruction from an address beyond ICCM address range.

5.13.1 Exception Stack Frame

Figure 13. Exception Frame Saved on the Stack for the #MC Exception



Note: The CPU pushes 2 additional 32-bit values on the stack as shown in Figure 13. Software reads these values in the exception handler to determine the address and nature of the access that generated the fault. When a fault occurs, the CPU always reports in the lowest address of a multi-byte data access or instruction fetch. Software should not alter the value of the reserved field.

Table 7. Exception Frame Stack Descriptions

Field	Description
W	Write Flag - 1 if the fault was caused by a write operation. 0 if the fault was caused by a read operation. This bit is only valid when the Instruction Flag is 0.
I	Instruction Flag - 1 if the fault was caused by an instruction fetch. 0 if the fault was not caused by an instruction fetch.

5.13.2 Exception Class

Abort.

5.13.3 Error Code

The CPU pushes two 32-bit words of error information for #MC as described in Figure 13.

Software must pop the error code from the stack before returning from the exception service routine. The stack pointer (ESP) must point to the EIP field of the stack frame before executing IRET.

5.13.4 Saved Instruction Pointer

The exception stack frame contains the EIP of the instruction executing at the time of the exception. The relationship between the EIP and the source of the #MC is undefined.

5.13.5 Program State Change

A program-state change does not accompany this exception, because the exception occurs before core executes the faulting instruction.

5.14 Exceptions 32-255 - User Defined Interrupts

The CPU generates a User Defined interrupt when:

- Software executes an INT instruction
- The CPU recognizes an external interrupt from the APIC

5.14.1 Exception Stack Frame

Figure 14. Exception Frame Saved on the Stack for External Interrupts

31 ... 0	
EFLAGS	ESP+8
0/Ignored	ESP+4
EIP	ESP

5.14.2 Exception Class

Interrupt.

5.14.3 Error Code

None.

5.14.4 Saved Instruction Pointer

The exception stack frame contains the EIP of the instruction following the INT instruction or the instruction following the instruction on which the external interrupt occurred.

5.14.5 Program State Change

A software process may resume on return from the interrupt handler without loss of continuity, provided the interrupt handler saves the state of the CPU before handling the interrupt and restores the CPU's state prior to a return.

5.15 Exception Ordering and Priority

This section describes the general ordering and prioritization of exception conditions by the CPU. At any given moment, the CPU will have multiple instructions in flight, each of which might generate a trap or fault. Simultaneously, the CPU also handles interrupts as well as machine check conditions. The CPU does not architecturally guarantee every aspect of exception processing, but follows general rules.

5.15.1 Trap and Fault Order

When considering only a single in-flight instruction, the CPU guarantees trap and fault order as follows. This is not prioritization per se, but the in-order sequence of possible events as an instruction progresses through the processor pipeline:

1. (Highest Priority) Machine Check Fault (#MC) (BUS_ERR) on code read
2. Invalid Opcode Fault (#UD)
3. Divide Error (#DE), INT instruction
4. (lowest priority) Machine Check Fault (#MC) (BUS_ERR) on data write

When two or more in flight instructions generate a trap in the same cycle, the exception from the oldest instruction (closest to retirement) takes priority.

5.15.2 Interrupts Versus Trap and Fault Order

When an external interrupt and a trap or fault are pending in the same cycle, the CPU uses the priority shown below to determine which event to service. Lettered sub-items within each priority level are also shown in priority order.

Note that the servicing an exception may itself trigger a fault condition, usually due to problems detected in the Interrupt Descriptor Table (IDT).

1. (Highest Priority) Hardware Reset and Errors
   a. RESET input
2. Exception Processing Unit (EPU) exceptions generated during active exception processing:
   a. Triple Fault
   b. Double Fault (#DF) (after #MC, #DE, #GP, #NP)
   c. General Protection Fault (#GP) on IDT length error
   d. Not-Present Fault (#NP)
   e. Machine Check Fault (#MC) on IDT read
3. Traps on the current instruction
   a. INT instruction
   b. Hardware Breakpoint
   c. Probe Mode Breakpoint
   d. EFLAGS.TF
4. Machine Check Fault (#MC) on BUS_ERR input asserted for a data write
5. Faults on the current instruction (see Section 5.15.1)
6. (Lowest Priority) Maskable hardware interrupts

5.16 Logical Algorithms

The CPU follows the algorithms shown in Figure 15, Figure 16 and Figure 17 for exception handling. For details on interrupt exit processing, refer to the IRET instruction in Section 8.34.

Figure 15. Hardware Operations Performed on Exception Entry

```
INPUT: Vector - Vector number of this exception, 0-255
INPUT: ErrVector - Vector number for IDT Errors, 0-255
INPUT: IDT - 1 = IDT Entry error, 0 = no IDT Entry error
INPUT: EXT - 1 = External interrupt, 0 = trap or fault. EXT = 1 implies INT = 0
INPUT: INT - 1 = INT instruction, 0 = not INT. INT = 1 implies EXT = 0
/* Inputs needed for #MC */
INPUT: Address - Faulting Address, if applicable
INPUT: I - 1 = Instruction fetch, 0 = not instruction fetch
INPUT: W - 1 = Data write, 0 = data read
/* Remember old state and switch to supervisor */
TempEFLAGS ← EFLAGS;
TempPM ← PM;
PM.U ← 0;
EFLAGS.TF ← 0;
IF ((Vector << 3) + 7) > IDTR.Limit THEN
    /* IDT error is new #GP. If already #DF, then triple fault */
    IF Vector = 8 THEN
        Triple Fault;
        DONE
    ENDIF
    /* If already #DE or #GP or #MC, then double fault */
    IF (Vector = 0) or (Vector = 13) or (Vector = 18) THEN
        #DF(ErrVector=0,IDT=0,EXT=0);
        DONE
    ENDIF
    #GP(ErrVector=Vector,IDT=1,EXT=EXT);
    DONE
ENDIF
DescAddr ← IDTR.Base + (Vector << 3);
/* Continued in Figure 16*/
```

Note: The algorithm continues in Figure 16 This algorithm is an architectural representation that does not reflect any particular hardware implementation

Figure 16. Hardware Operations Performed on Exception Entry Primarily Related to the IDT.P Bit (Continued from Figure 15)

```
/* Continued from Figure 15*/
    Desc ← Read(DescAddr);
    IF (Desc.P = 0) or ((Desc.Type ≠ TRAP GATE) and (Desc.Type ≠ INTERRUPT
    GATE) THEN
        /* IDT error is new #NP or #GP */
        IF Vector = 8 THEN /* If already #DF, then triple fault */
            Triple Fault;
            DONE
        ENDIF
        /* If already #DE or or #NP or #GP or #MC, then double fault */
        IF (Vector = 0) or (Vector = 11) or (Vector = 13) or (Vector = 18) or (Vector =
        24) THEN
            #DF(ErrVector=0,IDT=0,EXT=0);
            DONE
        ENDIF
        IF (Desc.P = 0) THEN
            #NP(ErrVector=Vector,IDT=1,EXT=EXT);
        ELSE
            #GP(ErrVector=Vector,IDT=1,EXT=EXT);
        ENDIF
        DONE
    ENDIF
    /* If INT instruction, check privilege */
    IF (INT = 1) and (Desc.U = 0) and (PM.U = 1) THEN
        #GP(ErrVector=Vector,IDT=0,EXT=0);
        DONE
    ENDIF
    IF Desc.Type = INTERRUPT GATE THEN
        EFLAGS.IF ← 0;
    ENDIF
    /* Continued in Figure 17*/
```

Note: The next figure, Figure 17, continues the algorithm beginning with exception handling from user mode illustration. This algorithm is an architectural representation that does not reflect any particular hardware implementation.

Figure 17. Hardware Operations Performed on Exception Entry from Supervisor Mode (Continued from Figure 16)

```
/* Continued from Figure 16*/
/* Exception entry from supervisor mode */
/* No stack switch, ESP update is all-or-nothing */
[ESP- 4] ← TempEFLAGS;
[ESP-8] ← TempPM;
IF (INT = 1) or (Vector = 1) THEN
    /* Trap, so IRET to next instruction */
    [ESP - 12] ← Next EIP;
ELSE
    /* Fault or interrupt, so IRET to current instruction */
    [ESP -12] ← EIP;
ENDIF
IF (Vector = 13) or (Vector = 8) THEN
    /* Push error code for #GP or #DF */
    [ESP - 16] ← Error Code(ErrVector,IDT,EXT);
    ESP ← ESP - 16;
ELSE
    IF (Vector = 18) or (Vector = 24) THEN
        /* Push error code and address for #MC */
        [ESP- 16] ← Error Code(I,TempPM.U,W);
        [ESP-20] ← Address;
        ESP ← ESP - 20;
    ELSE
        /* No error codes */
        ESP ← ESP - 12;
    ENDIF
ENDIF
EIP ← Desc.Address(31-0);
```

Note: This is an architectural representation that does not reflect any particular hardware implementation.

§ §

6.0 Reset

On a hardware reset, the CPU performs the initialization procedure shown in Figure 18. From end of reset to execution of the first instruction requires approximately 14 clock cycles.

Figure 18. Hardware Operations Performed on Reset

```
EIP ← 0;
ESP ← 0;
EFLAGS ← 0x2;
EAX ← 0;
EBX ← 0;
ECX ← 0;
EDX ← 0;
EBP ← 0;
ESI ← 0;
EDI ← 0;
PM.U ← 0;
ESP0 ← 0;
IDTR.Address ← 0;
IDTR.Limit ← 0;
```

6.1 Firmware Initialization Overview

The CPU resets into 32-bit physical addressing mode. At a minimum, the CPU requires firmware to initialize the stack pointer (ESP) and the Interrupt Descriptor Table (IDT).

Firmware created with C/C++ typically contains additional initialization overhead as required by the .elf format firmware image, such as clearing the .bss section.

6.2 Stack Initialization

Before other initialization, firmware should initialize the stack pointer. The stack grows downward in memory. Because a PUSH instruction decrements the stack pointer first, then stores data, firmware should initialize the stack pointer to the first 32-bit address after data RAM. Placing data in RAM above the stack is not recommended since stack underflow would result in a silent data corruption.

6.3 IDT Initialization

For exception handling, the CPU requires firmware to create an Interrupt Descriptor Table (IDT) and load the location of the table using the LIDT instruction. See Section 8.38 and Chapter 5.0. Each entry in the IDT consumes 8 bytes, with the first 32 entries reserved for processor generated traps and faults.

6.3.1 IDT Location

During exception processing, the Exception Processing Unit (EPU) performs one or more data reads (as opposed to code reads) from the IDT. Firmware may locate the IDT in code FLASH, data FLASH or SRAM. An easily identifiable IDT base address can help with debugging.

6.3.2 IDT Alignment

The CPU does not have alignment restrictions on the IDT. However, software should align the IDT on an 8 byte boundary to maximize efficiency.

§ §

7.0 APIC and IOAPIC

The CPU Advanced Programmable Interrupt Controller (APIC) controls external interrupt processing for the CPU and also provides a programmable timer. The APIC contains 2 main sub-modules: the I/O APIC (IOAPIC) and the Local APIC (LAPIC), each modeled on the x86 equivalent. The following sections describe each module in detail. This document uses APIC to refer to the interrupt controller as a whole, including both IOAPIC and LAPIC. Figure 19 shows an overview of the APIC.

7.1 Interrupt Vectors and Priorities

The CPU associates a vector number with each interrupt source. The APIC and core use the vector to determine interrupt priority as well as the IDT entry for the interrupt service routine address. The CPU uses 8 bit vector numbers, of which software programs the bottom 5 bits. The CPU reserves the low 32 vectors (0-31) for synchronous exceptions generated caused by software. External IOAPIC interrupts and the APIC Timer interrupt use vectors from 32 to 47.

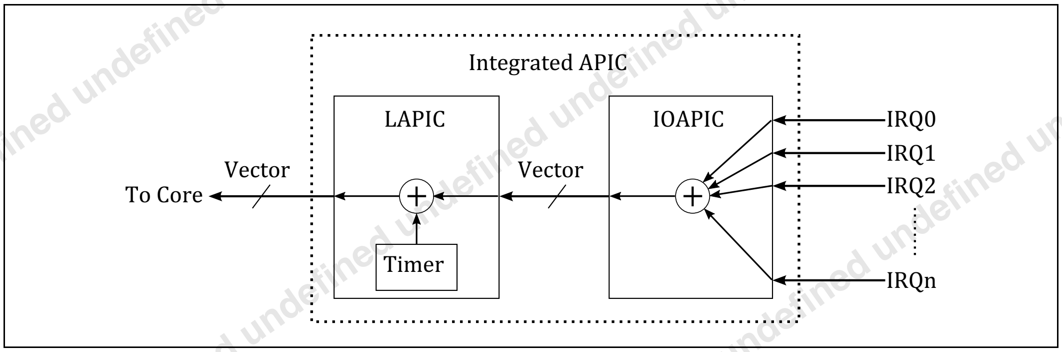
The larger the vector number, the higher the priority of the interrupt. Higher priority interrupts preempt lower priority interrupts. Lower priority interrupts do not preempt higher priority interrupts. The APIC holds the lower priority interrupts pending until the interrupt service routine for the high priority interrupt writes to the End of Interrupt (EOI) register. After an EOI write, the APIC asserts the next highest pending interrupt.

7.2 External Interrupts

This section describes the association of the external interrupts with processor interrupt vectors. The CPU provides 16 external interrupt sources as shown in Table 8.

The APIC Timer interrupt occurs at the vector value specified by software in the LVTTIMER register (Refer to Section 7.4.1, “Local Vector Table Timer Register (LVTTIMER)” on page 48). To avoid a conflict with external interrupts, the CPU reserves vector 45 for use by the APIC timer. Alternatively, software may program any vector value 32-47 for the APIC Timer if the external interrupt source is not in use.

Figure 19. Overview of the APIC that Integrates Both Local APIC and IOAPIC Functionality



Note: The APIC has 16 IRQ inputs.

Table 8. External Interrupt Sources and Associated Interrupt Vector

Vector	IDT Offset	Description
32	100h	GPIO
33	108h	I2C
34	110h	UART 0
35	118h	UART 1
36	120h	SPI Slave
37	128h	SPI Master
38	130h	Comparator
39	138h	ADC Command Complete
40	140h	ADC Mode Change Complete
41	148h	FLASH Command Complete
42	150h	Timer 0
43	158h	Timer 1
44	160h	Real-Time Clock
45	168h	APIC Timer
46	170h	Watch Dog Timer
47	178h	Security

Note: Interrupt priority increases with the vector number, ie. the security IREQ at Vector 47 has the highest priority.

7.3 Local APIC Registers

This section describes the memory-mapped registers implemented in the APIC. The base address for the Local APIC is FEE00000h and the memory range reserved for the Local APIC is FEE00000h to FEEFFFFFh. The CPU ignores reads or writes to reserved registers or fields. Refer to Table 9.

Table 9. Local APIC Memory Mapped Registers

Memory Mapped Address	Register Name	Access	Description
FEE00080h	TPR	R/W	Task Priority Register
FEE000A0h	PPR	RO	Process Priority Register
FEE000B0h	EOI	WO	End-of-Interrupt Register
FEE000F0h	SIVR	R/W	Spurious Interrupt Vector Register
FEE00110h	ISR	RO	In-Service Register, vectors 63-32
FEE00210h	IRR	RO	Interrupt Request Register, vectors 63-32
FEE00320h	LVTTIMER	R/W	Local Vector Table Timer Register
FEE00380h	ICR	R/W	Timer Initial Count Register
FEE00390h	CCR	RO	Timer Current Count Register

Note: All registers are 32-bits wide and have a reset value of 0, except the LVTTIMER Register which has a reset value of 00010000h.

7.3.1 Task Priority Register (TPR)

Address: FEE00080h

Software writes to this register with a vector number to set a priority threshold. The APIC will not deliver unmasked interrupts with a vector number lower than the TPR value. For example, a value of 0h allows all interrupts. A value of FFh disallows all interrupts.

Figure 20. Task Priority Register

31 ... 8	7 ... 0
0/Ignored	Vector

Note: Use this register to block low priority interrupts from interrupting the CPU. This register is read and writable.

7.3.2 Processor Priority Register (PPR)

Address: FEE000A0h

The APIC sets the Processor Priority Register to either to the highest priority pending interrupt in the ISR or to the current task priority, whichever is higher.

Figure 21. Processor Priority Register

31 ... 8	7 ... 0
0/Ignored	Vector

Note: Use this register to determine the priority at which the APIC is currently blocking interrupts. This register is read-only.

7.3.3 End-of-Interrupt Register (EOI)

Address: FEE000B0h

After an interrupt handler for any interrupt has completed servicing the interrupt request, the handler must write to this register before executing the IRET instruction at the end of the handler. Upon receipt of the EOI write, the local APIC clears the highest-priority ISR bit, which corresponds to the interrupt that was just serviced. The APIC ignores the value written to the EOI Register.

Figure 22. End-of-Interrupt Register

31–0
0/Ignored

Note: Use this register to tell the APIC when software completes interrupt processing. This register is write-only.

7.3.4 Spurious Interrupt Vector Register (SIVR)

Address: FEE000F0h

Software writes the vector used for spurious interrupts to the SIVR. The power-on default is 0xFF, but software may select any value from 20h to FFh.

Figure 23. Spurious Interrupt Vector Register

31–8	7–0
0/Ignored	Vector

Note: Use this register to handle the rare corner case of spurious interrupts. This register is read and writable.

A spurious interrupt occurs when an interrupt is pending, i.e. not yet acknowledged by the CPU and a write to the TPR register occurs with a new vector value greater than or equal to the pending interrupt vector. The APIC would normally disallow the pending interrupt, but since the interrupt signal is already asserted, the interrupt remains asserted until acknowledged.

However, in this special case the APIC generates this spurious vector number instead of the original vector number of the pending interrupt. After software acknowledges the spurious interrupt, the APIC does not set a status in the In-Service Register. Furthermore, the spurious interrupt handler is a simple stub containing only an IRET instruction. Software does not write to EOI for spurious interrupts since the APIC does not set a corresponding bit in the In-Service Register.

7.3.5 In-Service Register (ISR) Bits 47:32

Address: FEE00110h

The ISR tracks interrupts that have already requested service to the CPU but have not yet been acknowledged by software. The APIC set the bit in ISR after the CPU recognizes the corresponding interrupt. The APIC clears the bit in the ISR when software writes to the EOI register. Bit N corresponds to interrupt request N for interrupt vectors 32 to 47.

Figure 24. In-Service Register

31–16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved (0)	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32

Note: Each bit in the ISR corresponds to an in-service interrupt on the given vector number. Use this register to determine which interrupts the CPU is actively processing. This register is read only.

7.3.6 Interrupt Request Register (IRR) Bits 63:32

Address: FEE00210h

The IRR contains the active interrupt requests that have been accepted, but not yet dispatched to the CPU for servicing. When the local APIC accepts an interrupt, it sets the bit in the IRR that corresponds the vector of the accepted interrupt. When the CPU is ready to handle the next interrupt, the local APIC clears the highest priority IRR bit that is set and sets the corresponding ISR bit. The vector for the highest priority bit set in the ISR is then dispatched to the processor core for servicing.

Figure 25. Interrupt Request Register

31–16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved (0)	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32

Note: Each bit in the IRR corresponds to an interrupt on the given vector number that has not yet been dispatched to the CPU. Use this register to determine which interrupts are waiting for service. This register is read only.

7.4 Local APIC Timer

The Local APIC supports a timer. The timer runs at a constant rate regardless of clock and power state transitions in the CPU.

The following sections describe LAPIC registers pertaining to the timer.

7.4.1 Local Vector Table Timer Register (LVTTIMER)

Address: FEE00320h

The LVT Timer Register controls interrupt delivery when the APIC timer expires.

Figure 26. LVT Timer Register

31–18	17	16	15–8	7	6	5	4–0
0/Ignored	P	M	0/Ignored	0	0	1	Vector

P — Periodic Mode - Software sets this bit to operate the timer in periodic mode. In this mode, the timer automatically reloads the initial count value when the current count reaches zero. When

	this bit is clear, the timer operates in one-shot mode and does not automatically reload the count down value.
M	Mask - Software sets this bit to mask the timer interrupt. When this bit is clear, the timer generates an interrupt when the current count value reaches zero. When this bit is set, the timer does not generate an interrupt. On reset, the Mask bit is 1 which masks the interrupt.
Vector	Software writes this value to specify the interrupt vector used for timer interrupts. The LAPIC hard-codes bits 5,6 and 7 of the vector number as shown. The LAPIC ignores writes to the hard-coded bits.

Note: Use this register to initialize the timer's behavior and interrupt vector. This register is read and writable.

7.4.2 Initial Count Register (ICR)

Address: FEE00380h

The initial count for the timer. The timer counts down from this value to zero.

In periodic mode, the timer automatically reloads the Current Count Register (CCR) from the ICR when the count reaches 0. At this time, the APIC generates a timer interrupt to the CPU and the countdown repeats. If during the countdown process software writes to the ICR, counting restarts using the new initial count value. A write of 0 to the ICR effectively stops the local APIC timer, in both one-shot and periodic mode.

The LVT Timer Register determines the vector number delivered to the CPU when the timer count reaches zero. Software can use the mask flag in the LVT timer register to block the timer interrupt.

Figure 27. Local APIC Timer Initial Count Register

31 … 0
Initial Count

Note: Use this register to set the timer's duration. This register is read and writable.

7.4.3 Current Count Register (CCR)

Address: FEE00390h

The current count for the timer.

Figure 28. Local APIC Timer Current Count Register

31 … 0
Current Count

Note: Use this register to determine how many cycles remain before the timer expires. This register is read and writable.

7.5 IOAPIC Registers

The CPU implements an integrated IOAPIC to simplify design effort and reduce interrupt latency. Software uses the IOAPIC register interface to mask or unmask interrupt inputs and assign interrupt vector numbers. Software accesses the IOAPIC registers by an indirect addressing scheme using two memory mapped registers, IOREGSEL and IOWIN. Only the IOREGSEL and IOWIN registers are directly accessible in the memory address space. To reference an IOAPIC register, software writes to IOREGSEL with a value specifying the indirect IOAPIC register to be accessed. Software then reads or writes the IOWIN register for the desired data from/to the IOAPIC register specified by bits [7:0] of the IOREGSEL register. Software must access the IOWIN register as a dword quantity.

The IOREGSEL register retains the last value written by software. Software may repeatedly access the one IOAPIC register with IOWIN without rewriting IOREGSEL.

Table 10 list the memory mapped registers of the IOAPIC. The IOAPIC ignores reads from or writes to reserved registers or fields.

Note: Appendix B provides examples of C- code to interact with the IOAPIC.

Table 10. IOAPIC Memory Mapped Registers

Memory Mapped Address	Register Name	Access	Description
FEC00000h	IOREGSEL	R/W	IOAPIC Register Select (index)
FEC00010h	IOWIN	R/W	IOAPIC Register Windows (data)

Note: All registers are 32-bits wide and have a reset value of 0.

Table 11. IOAPIC Memory Mapped Registers

Register Index	Register Name	Access	Description
10h	IOREDTBL 0 [31:0]	R/W	Redirection Entry 0 low
12h	IOREDTBL 1 [31:0]	R/W	Redirection Entry 1 low
—	—	—	—
10h + 2*N*	IOREDTBL *N* [31:0]	R/W	Redirection Entry *N* low
—	—	—	—
2Eh	IOREDTBL 15 [31:0]	R/W	Redirection Entry 15 low

Note: All registers are 32-bits wide and have a reset value of 0.

7.6 IOAPIC Redirection Entry Registers

For each external interrupt source, software must program the corresponding IOAPIC Redirection Entry Register to set the Mask bit to enable or disable the interrupt. Figure 29 shows the format of the Redirection Entry Register.

Figure 29. Format of The IOAPIC Redirection Entry Registers

63 … 32
0/Ignored

31 … 17	16	15	14 … 0
0/Ignored	M	T	0/Ignored

M	Mask - Software sets this bit to mask the interrupt signal and prevent the IOAPIC from delivering the interrupt. The IOAPIC ignores interrupts signaled on a masked interrupt pin and does not deliver nor hold the interrupt pending. Changing the mask bit from unmasked to masked after the APIC accepts the interrupt has no effect on that interrupt. This behavior is identical to the case where the device withdraws an interrupt before the APIC posts that interrupt to the processor. Software must handle the case where it sets the mask bit after the APIC accepts the interrupt, but before the CPU processes that interrupt. When this bit is 0, the IOAPIC does not mask the interrupt and results in the eventual delivery of the interrupt. The CPU sets the M bit on reset such that all interrupts are masked.
T	Trigger - Software sets this bit to configure the interrupt signal as level sensitive. Software clears this bit to configure the interrupt signal as edge sensitive.

Note: Use these registers to enable or disable specific IRQ's. Software must write 0 to reserved bits.

7.7 Edge/Level Triggered Interrupts

The IOAPIC supports software configuration of edge or level triggered interrupts. Software must set the T bit in the IORDTBL register as appropriate for the interrupt input.

7.8 Interrupt Polarity

The IOAPIC does not support software configuration of interrupt polarity. Designers must fix the polarity in hardware as appropriate for the source of each interrupt.

§ §

8.0 Instruction Set

The CPU uses variable length instructions which provide the most commonly used integer operations used by C/C++ compilers. The shortest instruction is 1 byte and the longest instruction is 12 bytes. The CPU does not impose address alignment restrictions on instructions.

Note: The CPU supports a subset of the IA- 32 instruction set. Most instructions are machine code compatible with IA-32.

8.1 Intel® Quark™ microcontroller D1000 CPU Instructions

The Intel® Quark™ microcontroller D1000 CPU instruction set was selected using the following criteria:

- Integer instruction
- Used by C compilers
- No microcode required
- Low gate count

In addition, the instruction set includes several instructions necessary to support operating systems, e.g. LIDT.

8.2 Instruction Prefixes

The CPU supports two instruction prefixes that may be applied to most arithmetic and move type instructions. Refer to Table 12. The description for each instruction specifically states if an instruction prefix may be applied to that instruction.

Table 12. Instruction Prefix Bytes

Prefix Byte (hex)	Description
66	16-bit Operand Size

8.2.1 16-bit Operand Override

The 16-bit Operand Size Override prefix (66h) changes the logical width of an operation from 32-bits to 16-bits for most ALU and move type instructions. The description for each instruction specifically lists the opcodes that allow the 66h prefix in the instruction's opcode table. Specifying the 66h prefix multiple times for the same instruction results in a Invalid Opcode Fault (#UD).

8.3 Addressing Modes

The CPU supports many addressing modes in flat (non-segmented) memory. Specifically, these addressing modes are:

- Displacement (also called Absolute)
- Base (also called Indirect)
- Base + Displacement
- (Index * Scale) + Displacement
- Base + Index + Displacement
- Base + (Index * Scale) + Displacement

8.4 Instruction Format

The machine code format of Intel® Quark™ microcontroller D1000 CPU instructions is identical to IA-32.

Figure 30. The CPU Instruction Format Exactly Follows IA-32 Encoding

Prefixes 0-2 bytes	Opcode 1-2 bytes	ModR/M 1 byte	SIB 1 byte	Displacement 1,2 or 4 bytes	Immediate 1,2 or 4 bytes

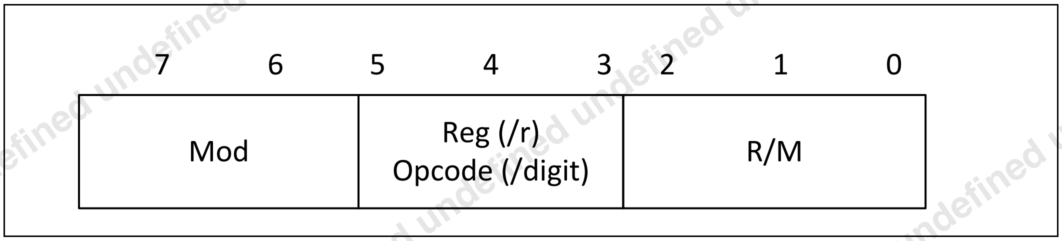
Note: All instructions use Opcode and require the other fields only as needed.

8.5 ModR/M Format

Many instructions that refer to an operand in memory have an addressingform 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 Mod field is the two most significant bits of the ModR/M value.
- The Reg/Opcode field specifies either a register number or three more bits of opcode information. The purpose of the Reg/Opcode field depends on the particular instruction.
- The R/M field can specify a register as an operand or can be combined with the Mod field to encode an addressing mode. Sometimes, certain combinations of the Mod field and the R/M field is used to express opcode information for some instructions. See Figure 31 for the bit format of the ModR/M byte.

Figure 31. Structure of the ModR/M Byte



Note: Bits 5-3 represent either a register selection (/r) or 3 additional opcode bits (/digit). Refer to Table 13 for more information.

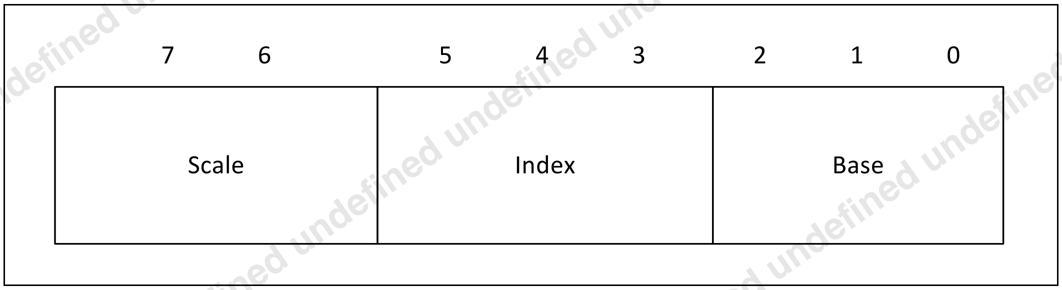
8.6 SIB Format

Certain encodings of the ModR/M byte require a second addressing byte specifying a Scale-Index-Base (SIB). The base-plus-index and no-base-plus-index forms 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 Figure 32 for the bit format of the SIB byte.

Figure 32. Structure of the Scale-Index- Base (SIB) Byte



Note: Refer to Table 14 for more information on SIB Byte.

8.7 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte or the SIB byte if one is present. A displacement operand, if present, can be 1, 2, or 4 bytes. An immediate operand, if present, follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

8.8 Opcode Column in Instruction Description

The Opcode column in the following sections 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.

/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	A 1-byte (cb), 2-byte (cw) or 4-byte (cd) value following the opcode. This value is used to specify a code offset relative to the address of the first byte past the end of the instruction.
ib, iw, id	A 1-byte (ib), 2-byte (iw) or 4-byte (id) 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 and double words are given with the low-order byte first.

Table 13. Addressing Modes Specified with the ModR/M Byte

r8(/r) r16(/r) r32(/r) Extended Opcode (/digit) REG (binary)			AL AX EAX 0 000	CL CX ECX 1 001	DL DX EDX 2 010	BL BX EBX 3 011	AH SP ESP 4 100	CH BP EBP 5 101	DH SI ESI 6 110	BH DI EDI 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]	00	001	01	09	11	19	21	29	31	39
[EDX]	00	010	02	0A	12	1A	22	2A	32	3A
[EBX]	00	011	03	0B	13	1B	23	2B	33	3B
SIB	00	100	04	0C	14	1C	24	2C	34	3C
[Disp32]	00	101	05	0D	15	1D	25	2D	35	3D
[ESI]	00	110	06	0E	16	1E	26	2E	36	3E
[EDI]	00	111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8	01	001	41	49	51	59	61	69	71	79
[EDX]+disp8	01	010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8	01	011	43	4B	53	5B	63	6B	73	7B
SIB+disp8	01	100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8	01	101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8	01	110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8	01	111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32	10	001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32	10	010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32	10	011	83	8B	93	9B	A3	AB	B3	BB
SIB+disp32	10	100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32	10	101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32	10	110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32	10	111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL	11	001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL	11	010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BH	11	011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH	11	100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH	11	101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH	11	110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH	11	111	C7	CF	D7	DF	E7	EF	F7	FF

Note: Rows with SIB indicate that a Scale- Indexed-Base byte follows the ModR/M byte in the instruction encoding. Refer to Table 14 for information on the SIB format.

Table 14. Addressing Modes Specified with the SIB Byte

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

Note: SIB byte sometimes follows the ModR/M byte in the instruction encoding. A Base encoding of 5 (101b) shown as the [*] column is a special case. The effective address for SIB Base=5 depends on the MOD field of the ModR/M byte as shown in Table 15.

Table 15. Addressing Modes Specified with the SIB Byte for Base Encoding of 5 (101b)

MOD bits (Table 13)			00	01	10
SIB Byte (Table 14)	SS	Index	Effective Address		
05	00	000	[Disp32+EAX]	[Disp8+EBP+EAX]	[Disp32+EBP+EAX]
0D	00	001	[Disp32+ECX]	[Disp8+EBP+ECX]	[Disp32+EBP+ECX]
15	00	010	[Disp32+EDX]	[Disp8+EBP+EDX]	[Disp32+EBP+EDX]
1D	00	011	[Disp32+EBX]	[Disp8+EBP+EBX]	[Disp32+EBP+EBX]
25	00	100	[Disp32]	[EBP+Disp8]	[EBP+Disp32]
2D	00	101	[Disp32+EBP]	[Disp8+EBP+EBP]	[Disp32+EBP+EBP]
35	00	110	[Disp32+ESI]	[Disp8+EBP+ESI]	[Disp32+EBP+ESI]
3D	00	111	[Disp32+EDI]	[Disp8+EBP+EDI]	[Disp32+EBP+EDI]
45	01	000	[Disp32+EAX*2]	[Disp8+EBP+EAX*2]	[Disp32+EBP+EAX*2]
4D	01	001	[Disp32+ECX*2]	[Disp8+EBP+ECX*2]	[Disp32+EBP+ECX*2]
55	01	010	[Disp32+EDX*2]	[Disp8+EBP+EDX*2]	[Disp32+EBP+EDX*2]
5D	01	011	[Disp32+EBX*2]	[Disp8+EBP+EBX*2]	[Disp32+EBP+EBX*2]
65	01	100	[Disp32]	[EBP+Disp8]	[EBP+Disp32]
6D	01	101	[Disp32+EBP*2]	[Disp8+EBP+EBP*2]	[Disp32+EBP+EBP*2]
75	01	110	[Disp32+ESI*2]	[Disp8+EBP+ESI*2]	[Disp32+EBP+ESI*2]
7D	01	111	[Disp32+EDI*4]	[Disp8+EBP+EDI*4]	[Disp32+EBP+EDI*4]
85	10	000	[Disp32+EAX*4]	[Disp8+EBP+EAX*4]	[Disp32+EBP+EAX*4]
8D	10	001	[Disp32+ECX*4]	[Disp8+EBP+ECX*4]	[Disp32+EBP+ECX*4]
95	10	010	[Disp32+EDX*4]	[Disp8+EBP+EDX*4]	[Disp32+EBP+EDX*4]
9D	10	011	[Disp32+EBX*4]	[Disp8+EBP+EBX*4]	[Disp32+EBP+EBX*4]
A5	10	100	[Disp32]	[EBP+Disp8]	[EBP+Disp32]
AD	10	101	[Disp32+EBP*4]	[Disp8+EBP+EBP*4]	[Disp32+EBP+EBP*4]
B5	10	110	[Disp32+ESI*4]	[Disp8+EBP+ESI*4]	[Disp32+EBP+ESI*4]
BD	10	111	[Disp32+EDI*4]	[Disp8+EBP+EDI*4]	[Disp32+EBP+EDI*4]
C5	11	000	[Disp32+EAX*8]	[Disp8+EBP+EAX*8]	[Disp32+EBP+EAX*8]
CD	11	001	[Disp32+ECX*8]	[Disp8+EBP+ECX*8]	[Disp32+EBP+ECX*8]
D5	11	010	[Disp32+EDX*8]	[Disp8+EBP+EDX*8]	[Disp32+EBP+EDX*8]
DD	11	011	[Disp32+EBX*8]	[Disp8+EBP+EBX*8]	[Disp32+EBP+EBX*8]
E5	11	100	[Disp32]	[EBP+Disp8]	[EBP+Disp32]
ED	11	101	[Disp32+EBP*8]	[Disp8+EBP+EBP*8]	[Disp32+EBP+EBP*8]
F5	11	110	[Disp32+ESI*8]	[Disp8+EBP+ESI*8]	[Disp32+EBP+ESI*8]
FD	11	111	[Disp32+EDI*8]	[Disp8+EBP+EDI*8]	[Disp32+EBP+EDI*8]

Note: The two MOD bits that select the column are the MOD field of the preceding ModR/M byte. Only MOD bit combinations 00, 01 and 10 allow a SIB byte.

8.9 Instruction Column in Instruction Description

The Instruction column gives the syntax of the instruction statement as it could appear in an assembly program. Table 16 provides a list of the symbols used to represent operands in the instruction statements.

Table 16. Instruction Column Details

Instruction	Description
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	A relative address in the range from 32768 bytes before the end of the instruction to 32767 after the end of the instruction. The rel16 symbol applies to instructions with an operand size attribute of 16 bits.
rel32	A relative address in the range from 231 bytes before the end of the instruction to 231?1 after the end of the instruction. The rel32 symbol applies to instructions with an operand size attribute of 32 bits.
r8	One of the general-purpose byte registers: AL, CL, DL, BL, AH, CH, DH, or BH.
r16	One of the general-purpose word registers: AX, CX, DX, BX, BP, SI or DI.
r32	One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI or EDI.
maddr8	An absolute address (32-bit) of a byte in memory.
maddr16	An absolute address (32-bit) of a 16-bit word in memory.
madd32	An absolute address (32-bit) of a 32-bit dword in memory.
imm8	An immediate byte value. The imm8 symbol is a signed number between -128 and +127 inclusive. 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. This is a number between 2^31-1 and -2^31 inclusive.
r/m8	A byte operand that is either the contents of a byte general purpose register (AL, CL, DL, BL, AH, CH, DH, BH) or a byte from memory. The contents of memory are found at the address provided by the effective address computation.
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.
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.
m16&32	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 LIDT and SIDT instructions use the m16&32 operand.

8.10 Operation Section

The Operation section contains an algorithm description 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 bold-face keywords, such as: IF, THEN, ELSE and END for an if statement; or CASE... OF for a case statement.
- Early termination of the algorithm is indicated by DONE. Otherwise the algorithm runs to the end of the listing.
- 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, [EDI] indicates the contents of the location whose address is in register EDI.
- 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 bits, or from the ESI register if the address-size attribute is 32 bits.
- A $\leftarrow$ 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 than, less than, 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 operator 'and' is a boolean and returning true or false.
- The operator 'or' is a boolean or returning true or false.
- The operator AND performs a bitwise logical AND operation.
- The operator NOT performs a bitwise logical inversion operation. A 0 bit becomes a 1 and a 1 becomes a 0.
- The operator OR performs a bitwise logical OR operation.
- The operator XOR performs a bitwise logical Exclusive-OR operation.
- The expression Carry() represents a carry or borrow out of the most significant bit of the unsigned result of an instruction. Carry() is 1 for a carry or borrow out condition, 0 otherwise.
- The expression Zero() is 1 if the result of an instruction is zero, 0 otherwise.
- The expression Sign() is 1 if the most significant bit (the sign bit) of the result of an instruction is set, 0 otherwise.
- The expression Overflow() represents a carry or borrow out of the most significant bit of the signed result of an instruction. This condition occurs when the sign of both operands is the same but different than the sign of the result. See Table 17.
- The expression Sizeof() represents the number of bytes in specified operand.

Table 17. Behavior of the Overflow Flag (EFLAGS.OF) Bit After an Arithmetic Operation

Operands	Result	
Sign ± Sign	**Sign**	**OF**
0 + 0	0	0
0 + 0	1	1
0 + 1	0	0
0 + 1	1	0
1 + 0	0	0
1 + 0	1	0

Table 17. Behavior of the Overflow Flag (EFLAGS.OF) Bit After an Arithmetic Operation

Operands	Result	
Sign ± Sign	Sign	OF
1 + 1	0	1
1 + 1	1	0
0 – 0	0	0
0 – 0	1	0
0 – 1	0	0
0 – 1	1	1
1 – 0	0	1
1 – 0	1	0
1 – 1	0	0
1 - 1	1	0

Note: The operands and result have sign bits as shown. Overflow cases (OF=1) are shaded gray.

8.11 Operand Order

For instructions with two operands, the instruction descriptions show the operands Destination,Source order. For example, the ADD instruction:

```
ADD r/m32, r32
```

...describes the source operand (second operand) as r32 and the destination operand (first operand) as r/m32. Specific assembler tools may use a different format.

8.12 ADC - Add with Carry

Opcode	Instruction
10 /r	ADC r/m8, r8
66 11 /r	ADC r/m16, r16
11 /r	ADC r/m32, r32
12 /r	ADC r8, r/m8
66 13 /r	ADC r16, r/m16
13 /r	ADC r32, r/m32
14 ib	ADC AL, imm8
66 15 iw	ADC AX, imm16
15 id	ADC EAX, imm32
80 /2 ib	ADC r/m8, imm8
66 81 /2 iw	ADC r/m16, imm16

Opcode	Instruction
81 /2 id	ADC r/m32, imm32
66 83 /2 ib	ADC r/m16, imm8
83 /2 ib	ADC r/m32, imm8

Adds the first operand (DEST), the second operand (SRC) and the carry flag (EFLAGS.CF) and stores the result in the first (DEST) 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. Two memory operands cannot be used in one instruction. When an immediate value is used as an operand, the CPU sign extends the value 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 often part of a multi-byte or multi-word addition in which an ADC instruction follows an ADD instruction. In this case, the state of the EFLAGS.CF represents the carry from the preceding ADD. The addition operation treats EFLAGS.CF as an integer 1 or 0.

8.12.1 Operation

Figure 33. ADC Algorithm

```
Temp ← SignExtend (SRC);
DEST ← DEST + Temp + EFLAGS.CF;
EFLAGS.CF ← Carry(DEST);
EFLAGS.ZF ← Zero(DEST);
EFLAGS.SF ← Sign(DEST);
EFLAGS.OF ← Overflow(DEST);
```

8.12.2 Exceptions

#MP	If the destination is a memory address and is unwritable or the source is a memory address and is unreadable. To detect this condition, the Intel® Quark™ microcontroller D1000 CPU must be configured with a Memory Protection Unit.

8.13 ADD - Add

Opcode	Instruction
00 /r	ADD r/m8, r8
66 01 /r	ADD r/m16, r16
01 /r	ADD r/m32, r32

Opcode	Instruction
02 /r	ADD r8, r/m8
66 03 /r	ADD r16, r/m16
03 /r	ADD r32, r/m32
04 ib	ADD AL, imm8
66 05 iw	ADD AX, imm16
05 id	ADD EAX, imm32
80 /0 ib	ADD r/m8, imm8
66 81 /0 iw	ADD r/m16, imm16
81 /0 id	ADD r/m32, imm32
66 83 /0 iw	ADD r/m16, imm8 (sign extended)
83 /0 id	ADD r/m32, imm8 (sign extended)

Adds the first operand (DEST) and the second operand (SRC) and stores the result in the first (DEST) 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. Two memory operands cannot be used in one instruction. When an immediate value is used as an operand, the CPU sign extends the value to the length of the destination operand format.

The ADD 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. For all possible operations, the ADD instruction produces 9 possible flag combinations. Table 18 shows an example of each combination.

Table 18. All EFLAG Combinations After Executing ADD for Various 8-bit Operands

DEST			SRC			DEST + SRC			EFLAGS			
h	ud	d	h	ud	d	h	ud	d	OF	SF	ZF	CF
7F	127	127	0	0	0	7F	127	127	0	0	0	0
FF	255	-1	7F	127	127	7E	126	126	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	1	0
FF	255	-1	1	1	1	0	0	0	0	0	1	1
FF	255	-1	0	0	0	FF	255	-1	0	1	0	0
FF	255	-1	FF	255	-1	FE	254	-2	0	1	0	1
FF	255	-1	80	128	-128	7F	127	127	1	0	0	1
80	128	-128	80	128	-128	0	0	0	1	0	1	1
7F	127	127	7F	127	127	FE	254	-2	1	1	0	0

Note: The h, ud and d columns show hexadecimal, unsigned decimal and signed decimal values respectively. Operation for 16 and 32-bit operands follows the same pattern. ELFAGS combinations not shown in this table cannot be generated by ADD.

8.13.1 Operation

Figure 34. ADD Algorithm

```
1 DEST ← DEST + SRC;
2 EFLAGS.CF ← Carry(DEST);
3 EFLAGS.ZF ← Zero(DEST);
4 EFLAGS.SF ← Sign(DEST);
5 EFLAGS.OF ← Overflow(DEST);
```

8.13.2 Exceptions

#MP	If the destination is a memory address and is unwritable or the source is a memory address and is unreadable. To detect this condition, Intel® Quark™ microcontroller D1000 CPU must be configured with a Memory Protection Unit.

8.14 AND - Logical AND

Opcode	Instruction
20 /r	AND r/m8, r8
66 21 /r	AND r/m16, r16
21 /r	AND r/m32, r32
22 /r	AND r8, r/m8
66 23 /r	AND r16, r/m16
23 /r	AND r32, r/m32
24 ib	AND AL, imm8
66 25 iw	AND AX, imm16
25 id	AND EAX, imm32
80 /4 ib	AND r/m8, imm8
66 81 /4 iw	AND r/m16, imm16
81 /4 id	AND r/m32, imm32
66 83 /4 ib	AND r/m16, imm8 (sign extended)
83 /4 ib	AND r/m32, imm8 (sign extended)

Performs a bitwise AND operation on the first operand (DEST) and second operand (SRC) and stores the result in the first (DEST) operand. The source operand can be an immediate, register or memory location. The destination operand can be a register or a memory location. Two memory operands cannot be used in one instruction. The CPU sets each bit of the result to 1 if both corresponding bits of the first and second operands are 1. Otherwise, the CPU sets the bit to 0.

8.14.1 Operation

Figure 35. AND Algorithm

```
DEST ← DEST AND SRC;
EFLAGS.CF ← 0;
EFLAGS.ZF ← Zero(DEST);
EFLAGS.SF ← Sign(DEST);
EFLAGS.OF ← 0;
```

8.14.2 Exceptions

#MP	If the destination is a memory address and is unwritable or the source is a memory address and is unreadable. To detect this condition, CPU must be configured with a Memory Protection Unit

8.15 BSWAP - Byte Swap

Opcode	Instruction
0F C8+rd	BSWAP r32

Reverses the byte order of a 32-bit register and stores the result in the register. This instruction converts little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

8.15.1 Operation

Figure 36. BSWAP Algorithm

```
Temp ← DEST;
DEST[7:0] ← Temp[31:24];
DEST[15:8] ← Temp[23:16];
DEST[23:16] ← Temp[15:8];
DEST[31:24] ← Temp[7:0];
```

8.16 BT - Bit Test

Opcode	Instruction
66 0F A3	BT r/m16, r16
0F A3	BT r/m32, r32
66 0F BA /4 ib	BT r/m16, imm8
0F BA /4 ib	BT r/m32, imm8

Selects the bit in the first operand (BASE), at the bit-position designated by the second operand (OFFSET) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location. The bit offset operand can be a register or an immediate value.

The instruction takes the modulo 16 or 32 of the bit offset operand for 16 and 32 bit operands respectively. The CPU ignores the upper bits of the offset operand.

If the bit base operand is a memory address, then this operand specifies is the address of the byte containing bit 0 of the bit base.

8.16.1 Operation

Figure 37. BT Algorithm

```
IF Sizeof(BASE) = 2 THEN
    /* 16-bit offset range. */
    Temp ← 1 << OFFSET[3:0];
ELSE
    /* 32-bit offset range. */
    Temp ← 1 << OFFSET[4:0];
ENDIF
IF (BASE AND Temp) ≠ 0 THEN
    EFLAGS.CF ← 1;
ELSE
    EFLAGS.CF ← 0;
ENDIF
EFLAGS.SF ← Undefined;
EFLAGS.OF ← Undefined;
```

8.17 BTC - Bit Test and Complement

Opcode	Instruction
66 0F BA /7 ib	BTC r/m16, imm8
0F BA /7 ib	BTC r/m32, imm8
66 0F BB	BTC r/m16, r16
0F BB	BTC r/m32, r32

Selects the bit in the first operand (BASE), at the bit-position designated by the second operand (OFFSET) and stores the value of the bit in the CF flag, then complements the bit in the bit base. The bit base operand can be a register or a memory location. The bit offset operand can be a register or an immediate value.

The instruction takes the modulo 16 or 32 of the bit offset operand for 16 and 32 bit operands respectively. The CPU ignores the upper bits of the offset operand.

If the bit base operand is a memory address, then this operand specifies is the address of the byte containing bit 0 of the bit base.

8.17.1 Operation

Figure 38. BTC Algorithm

```
IF Sizeof(BASE) = 2 THEN
    /* 16-bit offset range. */
    Temp ← 1 << OFFSET[3:0];
ELSE
    /* 32-bit offset range. */
    Temp ← 1 << OFFSET[4:0];
ENDIF
IF (BASE AND Temp) ≠0 THEN
    EFLAGS.CF ← 1;
    BASE.Temp ← 0;
ELSE
    EFLAGS.CF ← 0;
    BASE.Temp ← 1;
ENDIF
EFLAGS.SF ← Undefined;
EFLAGS.OF ← Undefined;
```

8.18 BTR - Bit Test and Reset

Opcode	Instruction
66 0F B3	BTR r/m16, r16
0F B3	BTR r/m32, r32
66 0F BA /6 ib	BTR r/m16, imm8
0F BA /6 ib	BTR r/m32, imm8

Selects the bit in the first operand (BASE), at the bit-position designated by the second operand (OFFSET) and stores the value of the bit in the CF flag, then clears the bit in the bit base. The bit base operand can be a register or a memory location. The bit offset operand can be a register or an immediate value.

The instruction takes the modulo 16 or 32 of the bit offset operand for 16 and 32 bit operands respectively. The CPU ignores the upper bits of the offset operand.

If the bit base operand is a memory address, then this operand specifies is the address of the byte containing bit 0 of the bit base.

8.18.1 Operation

Figure 39. BTR Algorithm.

```
IF Sizeof(BASE) = 2 THEN
    /* 16-bit offset range. */
    Temp ← 1 << OFFSET[3:0];
ELSE
    /* 32-bit offset range. */
    Temp ← 1 << OFFSET[4:0];
ENDIF
IF (BASE AND Temp) ≠ 0 THEN
    EFLAGS.CF ← 1;
ELSE
    EFLAGS.CF ← 0;
ENDIF
BASE.Temp ← 0;
EFLAGS.SF ← Undefined;
EFLAGS.OF ← Undefined;
```

8.19 BTS - Bit Test and Set

Opcode	Instruction
66 0F AB	BTS r/m16, r16
0F AB	BTS r/m32, r32
66 0F BA /5 ib	BTS r/m16, imm8
0F BA /5 ib	BTS r/m32, imm8

Selects the bit in the first operand (BASE), at the bit-position designated by the second operand (OFFSET) and stores the value of the bit in the CF flag, then sets the bit in the bit base. The bit base operand can be a register or a memory location. The bit offset operand can be a register or an immediate value.

The instruction takes the modulo 16 or 32 of the bit offset operand for 16 and 32 bit operands respectively. The CPU ignores the upper bits of the offset operand.

If the bit base operand is a memory address, then this operand specifies is the address of the byte containing bit 0 of the bit base.

8.19.1 Operation

Figure 40. BTS Algorithm

```
IF Sizeof(BASE) = 2 THEN
    /* 16-bit offset range. */
    Temp ← 1 << OFFSET[3:0];
ELSE
    /* 32-bit offset range. */
    Temp ← 1 << OFFSET[4:0];
ENDIF
IF (BASE AND Temp) ≠0 THEN
    EFLAGS.CF ← 1;
ELSE
    EFLAGS.CF ← 0;
ENDIF
Base.Temp ← 1;
EFLAGS.SF ← Undefined;
EFLAGS.OF ← Undefined;
```

8.20 CALL - Call Procedure

Opcode	Instruction
E8 cd	CALL rel32
FF /2	CALL r/m32

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

The E8h opcode form specifies a 32-bit relative code offset from the end of the instruction. The FFh opcode form performs an indirect branch to the value contained at the effective address of the operand.

8.20.1 Operation

Figure 41. CALL Procedure using Relative Jump with Opcode E8 cd

```
1 ESP ← ESP - 4;
/* sizeof(CALL) is 5 */
2 [ESP] ← EIP + 5;
3 EIP ← EIP + cd;
```

Figure 42. CALL Procedure using Absolute Address with Opcode FF /2

```
1 ESP ← ESP - 4;
/* sizeof(CALL) varies */
2 [ESP] ← EIP + Sizeof(CALL);
3 EIP ← DEST;
```

Note: The DEST value of the jump is specified by r/m32.

8.21 CBW/CWDE - Convert Byte to Word/Word to Doubleword

Opcode	Instruction
66 98	CBW AX
98	CWDE EAX

Doubles the size of the AL or AX register by means of sign extension and stores the result in the register AX or EAX respectively. The CBW instruction copies the sign (bit 7) of the value in the AL register into every bit position in the AH register. The CWDE instruction copies the sign (bit 15) of the value in the AX register into the high 16 bits of the EAX register.

The CBW instruction can be used to produce a word dividend from a byte before byte division. The CWDE instruction can be used to produce a doubleword dividend from a word before word division.

8.21.1 Operation

Figure 43. CBW Algorithm

```
1 AH[7:0] ←  AL[7];
```

Figure 44. CWDE Algorithm

```
1 EAX[31:16] ←  AX[15];
```

8.22 CLC - Clear Carry Flag

Opcode	Instruction
F8	CLC

Note: Clears the CF flag in the EFLAGS register.

8.22.1 Operation

Figure 45. CLC Algorithm

```
1 EFLAGS.CF ←  0;
```

8.23 CLI - Clear Interrupt Flag

Opcode	Instruction
FA	CLI

Note: CLI clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts.

8.23.1 Operation

Figure 46. CLI Algorithm

```
1 EFLAGS.IF ← 0;
```

8.24 CMC - Complement Carry Flag

Opcode	Instruction
F5	CMC

Note: Complements the CF flag in the EFLAGS register.

8.24.1 Operation

Figure 47. CMC Algorithm

```
1 EFLAGS.CF ← NOT EFLAGS.CF;
```

8.25 CMP - Compare Two Operands

Opcode	Instruction
38 /r	CMP r/m8, r8
66 39 /r	CMP r/m16, r16
39 /r	CMP r/m32, r32
3A /r	CMP r8, r/m8
66 3B /r	CMP r16, r/m16
3B /r	CMP r32, r/m32
3C ib	CMP AL, imm8
66 3D iw	CMP AX, imm16
3D id	CMP EAX, imm32
80 /7 ib	CMP r/m8, imm8
66 81 /7 iw	CMP r/m16, imm16
81 /7 id	CMP r/m32, imm32
66 83 /7 ib	CMP r/m16, imm8
83 /7 ib	CMP r/m32, imm8

Compares the first operand (SRC1) with the second operand (SRC2) and sets the status flags in the EFLAGS register according to the result. The CPU performs the comparison by subtracting SRC2 from SRC1 and then setting the status flags in the same manner as the SUB instruction, but without storing the result.

When the second operand is an immediate value, the CPU sign extends the value to the length of the first operand (SRC1).

For all possible comparisons, the CMP instruction produces 7 possible flag combinations. Table 19 shows an example of each.

Table 19. All EFLAG Combinations After Executing CMP for Various 8-bit Operands

SRC1			SRC2			SRC1-SRC2			EFLAGS			
h	ud	d	h	ud	d	h	ud	d	CF	SF	ZF	CF
FF	255	-1	FE	254	-2	1	1	1	0	0	0	0
7E	126	126	FF	255	-1	7F	127	127	0	0	0	1
FF	255	-1	FF	255	-1	0	0	0	0	0	1	0
FF	255	-1	7F	127	127	80	128	-128	0	1	0	0
FE	254	-2	FF	255	-1	FF	255	-1	0	1	0	1
FE	254	-2	7F	127	127	7F	127	127	1	0	0	0
7F	127	127	FF	255	-1	80	128	-128	1	1	0	1

The h, ud and d columns show hexadecimal, unsigned decimal and signed decimal values respectively. Operation for 16 and 32-bit operands follows the same pattern. ELFAGS combinations not shown in this table cannot be generated by CMP.

8.25.1 Operation

Figure 48. CMP Algorithm

```
Temp ← SignExtend (SRC2);
Temp ← SRC1 - Temp;
EFLAGS.CF ← Carry(Temp);
EFLAGS.OF ← Overflow(Temp);
EFLAGS.SF ← Sign(Temp);
EFLAGS.ZF ← Zero(Temp);
```

8.26 CWD/CDQ - Convert to Doubleword or Quadword

Opcode	Instruction
66 99	CWD DX:AX
99	CDQ EDX:EAX

Doubles the size of the AX or EAX register by means of sign extension and stores the result in the register DX:AX or EDX:EAX respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division.

Note: The GNU objdump utility reports the CDQ instruction as CLTD

8.26.1 Operation

Figure 49. CWD Algorithm

```
1 DX[15:0] ← AX[15];
```

Figure 50. CDQ Algorithm

```
1 EDX[31:0] ← EAX[31];
```

8.27 DEC - Decrement by 1

Opcode	Instruction
66 48	DEC AX
66 49	DEC CX
66 4A	DEC DX
66 4B	DEC BX
66 4D	DEC BP
66 4E	DEC SI
66 4F	DEC DI
48	DEC EAX
49	DEC ECX
4A	DEC EDX
4B	DEC EBX
4C	DEC ESP
4D	DEC EBP
4E	DEC ESI
4F	DEC EDI
FE /1	DEC r/m8
66 FF /1	DEC r/m16
FF /1	DEC r/m32

Subtracts 1 from the operand (DEST), while preserving the state of the CF flag. The CPU updates the OF, SF and ZF flags according to the result.

8.27.1 Operation

Figure 51. DEC Algorithm

```
1 DEST ← DEST - 1;
2 EFLAGS.OF ← Overflow(DEST);
3 EFLAGS.SF ← Sign(DEST);
4 EFLAGS.ZF ← Zero(DEST);
```

8.28 DIV - Unsigned Divide

Opcode	Instruction
F6 /6	DIV r/m8
66 F7 /6	DIV r/m16
F7 /6	DIV r/m32

Divides the unsigned value in the AX, DX:AX or EDX:EAX registers (dividend) by the source operand (divisor) and stores the result in the AX, 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). The CPU truncates (chops) non-integral results towards 0.

The remainder is always less than the divisor in magnitude. The CPU indicates overflow with the #DE (divide error) exception rather than with the CF flag.

8.28.1 Exceptions

#DE If the source operand (divisor) is 0.

#DE If the quotient is too large for the designated register.

8.29 HLT - Halt

Opcode	Instruction
F4	HLT

Stops instruction execution and places the CPU in a HALT state. An enabled interrupt, a debug exception or the RESET signal will resume execution. If an interrupt is used to resume execution after a HLT instruction, the saved instruction pointer (EIP) in the interrupt stack frame points to the instruction following the HLT instruction.

8.30 IDIV - Signed Divide

Opcode	Instruction
F6 /7	IDIV r/m8
66 F7 /7	IDIV r/m16
F7 /7	IDIV r/m32

8.30.1 Exceptions

#DE If the source operand (divisor) is 0.

#DE If the quotient is too large for the designated register.

8.31 IMUL - Signed Multiply

Opcode	Instruction
66 0F AF /r	IMUL r16, r/m16
0F AF /r	IMUL r32, r/m32
66 6B /r ib	IMUL r16, r/m16, imm8
6B /r ib	IMUL r32, r/m32, imm8
66 69 /r iw	IMUL r16, r/m16, imm16
69 /r id	IMUL r32, rr/m32, imm32
F6 /5	IMUL r/m8
66 F7 /5	IMUL r/m16
F7 /5	IMUL r/m32

8.31.1 Description

Performs a signed 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 AX, DX:AX or EDX:EAX depending on the size of the operand. The high-order bits of the product are contained in register AH, DX, or EDX, respectively. 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 20.

Table 20. Results of the MUL Instruction

Opcode	Operand Size(bits)	Source 1	Source 2	Destination
F6 /5	8	AL	r/m8	AX
66 F7 /5	16	AX	r/m16	DX:AX
F7 /5	32	EAX	r/m32	EDX:EAX

8.31.2 Operation

Figure 52. IMUL Algorithm

```
IF Sizeof(SRC) = 1 THEN
    AX ← AL * SRC;
    IF AH = 0 THEN
        EFLAGS.CF ← 0;
        EFLAGS.OF ← 0;
    ELSE
        EFLAGS.CF ← 1;
        EFLAGS.OF ← 1;
    ENDIF
ELSE
    IF Sizeof(SRC) = 2 THEN
        DX:AX ← AX * SRC;
        IF DX = 0 THEN
            EFLAGS.CF ← 0;
            EFLAGS.OF ← 0;
        ELSE
            EFLAGS.CF ← 1;
            EFLAGS.OF ← 1;
        ENDIF
    ELSE
        EDX:EAX ← EAX * SRC;
        IF EDX = 0 THEN
            EFLAGS.CF ← 0;
            EFLAGS.OF ← 0;
        ELSE
            EFLAGS.CF ← 1;
            EFLAGS.OF ← 1;
        ENDIF
    ENDIF
ENDIF
EFLAGS.SF ← Undefined;
EFLAGS.ZF ← Undefined;
```

8.32 INC - Increment by 1

Opcode	Instruction
66 40	INC AX
66 41	INC CX
66 42	INC DX
66 43	INC BX
66 45	INC BP
66 46	INC SI
66 47	INC DI
40	INC EAX
41	INC ECX
42	INC EDX
43	INC EBX
44	INC ESP
45	INC EBP
46	INC ESI
47	INC EDI
FE /0	INC r/m8
66 FF /0	INC r/m16
FF /0	INC r/m32

Adds 1 to the operand (DEST), while preserving the state of the CF flag. The CPU updates the OF, SF and ZF flags according to the result.

8.32.1 Operation

Figure 53. INC Algorithm.

```
1 DEST ← DEST + 1;
2 EFLAGS.OF ← Overflow(DEST);
3 EFLAGS.SF ← Sign(DEST);
4 EFLAGS.ZF ← Zero(DEST);
```

8.33 INT - Call to Interrupt Procedure

Opcode	Instruction
CC	INT3
CD ib	INT imm8

8.33.1 Description

The INT instruction generates a trap to the exception handler specified with the source operand. The CPU pushes the next EIP on the interrupt stack frame because INT is a trap type exception. A subsequent IRET instruction thus returns to the next instruction after the INT. If the INT instruction causes one of the following fault conditions, the CPU treats the INT as a fault and not a trap. In the faulting case, the CPU pushes the EIP of the INT instruction itself. A subsequent IRET will then re-execute the faulting INT.

8.33.2 Exceptions

#GP	If the destination address is outside the IDT limit.

8.34 IRET - Interrupt Return

Opcode	Instruction
CF	IRET

8.34.1 Description

The IRET instruction returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software generated interrupt.

8.34.2 Operation

Figure 54. IRET Algorithm

```
tempEIP ←  [ESP];
tempPM ←  [ESP+4];
tempEFLAGS ←  [ESP+8];
ESP ←  ESP + 12;
EFLAGS ←  tempEFLAGS;
PM ←  tempPM;
EIP ←  tempEIP;
```

8.35 Jcc - Jump if Condition is Met

Opcode	Instruction
70 cb	JO rel8
71 cb	JNO rel8
72 cb	JB rel8
73 cb	JAE rel8
74 cb	JE rel8
75 cb	JNE rel8
76 cb	JBE rel8
77 cb	JA rel8
78 cb	JS rel8
79 cb	JNS rel8
7C cb	JL rel8
7D cb	JGE rel8
7E cb	JLE rel8
7F cb	JG rel8
0F 80 cd	JO rel32
0F 81 cd	JNO rel32
0F 82 cd	JB rel32
0F 83 cd	JAE rel32
0F 84 cd	JE rel32
0F 85 cd	JNE rel32
0F 86 cd	JBE rel32
0F 87 cd	JA rel32
0F 88 cd	JS rel32
0F 89 cd	JNS rel32
0F 8C cd	JL rel32
0F 8D cd	JGE rel32
0F 8E cd	JLE rel32
0F 8F cd	JG rel32

The Jcc instructions conditionally jump depending on the state of one or more of the status flags in the EFLAGS register: CF, OF, SF and ZF. If the flags match the specified condition, execution jumps to the target instruction specified by the destination operand. If the flags do not match the specified condition, the jump is not performed and execution continues with the instruction following the Jcc instruction.

The destination operand specifies the target instruction as a signed relative offset from the address of the next byte after Jcc instruction.

Table 21. Common Aliases for Jcc Instructions

Original	Aliases
JA	JNBE
JAE	JNB
JB	JC, JNAE
JBE	JNA
JE	JZ
JG	JNLE
JGE	JNL
JL	JNGE
JLE	JNG
JNE	JNZ

Note: Assembler and disassembler tools may support these alternatives

Table 22. EFLAGS Condition Codes Associated with Each Conditional Jump Instruction

Name	Jump Condition	Description
JA	CF=0 and ZF=0	Jump if above.
JAE	CF=0	Jump if above or equal.
JB	CF=1	Jump if below
JBE	CF=1 or ZF=1	Jump if below or equal
JE	ZF=1	Jump if equal
JG	ZF=0 and SF=OF	Jump if greater
JGE	SF=OF	Jump if greater or equal
JL	SF≠ OF	Jump if less
JLE	ZF=1 and SF≠ OF	Jump if less or equal
JNE	ZF=0	Jump if not equal
JNO	OF=0	Jump if not overflow
JNS	SF=0	Jump if not sign
JO	OF=1	Jump if overflow
JS	SF=1	Jump if sign

8.36 JMP - Jump

Opcode	Instruction
EB cb	JMP rel8
E9 cd	JMP rel32
FF /4	JMP r/m32

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction to which the CPU jumps. This operand can be an immediate value, a general-purpose register, or a memory location. JMP instructions with opcodes EB and E9 specify a relative offset from the address of the byte following the JMP instruction.

8.37 LEA - Load Effective Address

Opcode	Instruction
66 8D /r	LEA r16, m32
8D /r	LEA r32, m32

8.37.1 Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address specified with one of the processors addressing modes. The destination operand is a general-purpose register.

Both forms of LEA compute the effective 32-bit address of the second operand. However, the form with the 66 prefix discards the upper 16-bit bits of the effective address and stores the lower 16-bit bits into the selected register.

8.37.2 Exceptions

#UD	If the source operand is not a memory location.

8.38 LIDT - Load Interrupt Descriptor Table Register

Opcode	Instruction
0F 01 /3	LIDT m

8.38.1 Description

The LIDT instruction loads the Interrupt Descriptor Table Register (IDTR) from a 6 byte memory structure defined in Figure 55. The operand m is the memory address of the structure.

Figure 55. IDTR Format

	31 — 16	15 — 0
04h	unused	IDT Address 31-16
00h	IDT Address 15-0	Byte Limit minus 1

Note: The LIDT instruction loads a pointer to this memory structure in the CPU's IDTR register.

Figure 56 shows example use of the LIDT instruction to setup an IDT with a full 256 entries.

Figure 56. Example Use of the LIDT Instruction to Setup an IDT with a Full 256 Entries

```
.extern my_idt # define non-local label

idtr_value: # Reference address of 6 byte IDTR

.word 0x07FF # 8*N - 1 for N entries in the IDT

.long my_idt # Address of start of IDT

start_of_code:

...

lidt idtr_value

...
```

8.38.2 Exceptions

#UD	If the source operand is not a memory location.
#UD	If the 66h prefix is used.

8.39 MOV - Move

Opcode	Instruction
88 /r	MOV r/m8, r8
66 89 /r	MOV r/m16, r16
89 /r	MOV r/m32, r32
8A /r	MOV r8, r/m8
66 8B /r	MOV r16, r/m16
8B /r	MOV r32, r/m32
A0	MOV AL, maddr8
66 A1	MOV AX, maddr16
A1	MOV EAX, maddr32
A2	MOV maddr8, AL
66 A3	MOV maddr16, AX
A3	MOV maddr32, EAX
B0	MOV AL, imm8
B1	MOV CL, imm8
B2	MOV DL, imm8
B3	MOV BL, imm8
B4	MOV AH, imm8
B5	MOV CH, imm8
B6	MOV DH, imm8
B7	MOV BH, imm8
66 B8	MOV AX, imm16
66 B9	MOV CX, imm16
66 BA	MOV DX, imm16
66 BB	MOV BX, imm16
66 BD	MOV BP, imm16
66 BE	MOV SI, imm16
66 BF	MOV DI, imm16
B8	MOV EAX, imm32
B9	MOV ECX, imm32
BA	MOV EDX, imm32
BB	MOV EBX, imm32
BC	MOV ESP, imm32
BD	MOV EBP, imm32
BE	MOV ESI, imm32
BF	MOV EDI, imm32
C6 /0	MOV r/m8, imm8
66 C7 /0	MOV r/m16, imm16
C7 /0	MOV r/m32, imm32

Copies the second operand (SRC) to the first operand (DEST). The source operand can be an immediate value, general-purpose register or memory location. The destination register can be a general purpose register or memory location. Both operands must be the same size, which can be a byte, a word (16-bit) or a doubleword (32-bit). MOV does not affect processor flags.

8.39.1 Operation

Figure 57. MOV Algorithm.

```
1 DEST ←   SRC;
```

8.40 MOVSX - Move with Sign-Extend

Opcode	Instruction
66 0F BE /r	MOVSX r16, r/m8
0F BE /r	MOVSX r32, r/m8
0F BF /r	MOVSX r32, r/m16

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. The size of the converted value depends on the operand-size attribute.

8.41 MOVZX - Move with Zero-Extend

Opcode	Instruction
66 0F B6 /r	MOVZX r16, r/m8
0F B6 /r	MOVZX r32, r/m8
0F B7 /r	MOVZX r32, r/m16

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

8.42 MUL - Unsigned Multiply

Opcode	Instruction
F6 /4	MUL r/m8
66 F7 /4	MUL r/m16
F7 /4	MUL r/m32

8.42.1 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 AX, DX:AX or EDX:EAX depending on the size of the operand. The high-order bits of the product are contained in register AH, DX, or EDX, respectively. 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 23.

Table 23. Results of the MUL Instruction

Opcode	Operand Size (bits)	Source 1	Source 2	Destination
F6 /4	8	AL	r/m8	AX
66 F7 /4	16	AX	r/m16	DX:AX
F7 /4	32	EAX	r/m32	EDX:EAX

8.42.2 Operation

Figure 58. MUL Algorithm

```
IF Sizeof(SRC) = 1 THEN
    AX ← AL * SRC;
    IF AH = 0 THEN
        EFLAGS.CF ← 0;
        EFLAGS.OF ← 0;
    ELSE
        EFLAGS.CF ← 1;
        EFLAGS.OF ← 1;
    ENDIF
ELSE
    IF Sizeof(SRC) = 2 THEN
        DX:AX ← AX * SRC;
        IF DX = 0 THEN
            EFLAGS.CF← 0;
            EFLAGS.OF ← 0;
        ELSE
            EFLAGS.CF ← 1;
            EFLAGS.OF ← 1;
        ENDIF
    ELSE
        EDX:EAX← EAX * SRC;
        IF EDX = 0 THEN
            EFLAGS.CF← 0;
            EFLAGS.OF ← 0;
        ELSE
            EFLAGS.CF ← 1;
            EFLAGS.OF ← 1;
        ENDIF
    ENDIF
ENDIF
EFLAGS.SF ← Undefined;
EFLAGS.ZF ← Undefined;
```

8.43 NEG - Two's Complement Negation

Opcode	Instruction
F6 /3	NEG r/m8
66 F7 /3	NEG r/m16
F7 /3	NEG r/m32

Replaces the value of the operand (DEST) 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.

8.43.1 Operation

Figure 59. NEG Algorithm

```
DEST ← 0 - DEST;
IF DEST = 0 THEN
    EFLAGS.CF ← 0;
ELSE
    EFLAGS.CF ← 1;
ENDIF
EFLAGS.OF ← Overflow(DEST);
EFLAGS.SF ← Sign(DEST);
EFLAGS.ZF ← Zero(DEST);
```

8.44 NOP - No Operation

Opcode	Instruction
90	NOP
66 90	NOP

This instruction performs no operation. NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register. The NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

8.45 NOT - One's Complement Negation

Opcode	Instruction
F6 /2	NOT r/m8
66 F7 /2	NOT r/m16
F7 /2	NOT r/m32

Performs a bitwise NOT operation on the operand (DEST) and stores the result to the operand. The operand is modified such that each 1 is set to 0 and each 0 is set to 1. The operand can be a register or a memory location. NOT does not affect processor flags.

8.45.1 Operation

Figure 60. NOT Algorithm.

```
DEST ← NOT DEST;
```

8.46 OR - Logical Inclusive OR

Opcode	Instruction
08 /r	OR r/m8, r8
66 09 /r	OR r/m16, r16
09 /r	OR r/m32, r32
0A /r	OR r8, r/m8
66 0B /r	OR r16, r/m16
0B /r	OR r32, r/m32
0C ib	OR AL, imm8
66 0D iw	OR AX, imm16
0D id	OR EAX, imm32
80 /1 ib	OR r/m8, imm8
66 81 /1 iw	OR r/m16, imm16
81 /1 id	OR r/m32, imm32
66 83 /1 ib	OR r/m16, imm8
83 /1 ib	OR r/m32, imm8

Performs a bitwise inclusive OR operation between the first operand (DEST) and second operand (SRC) and stores the result in the destination operand. 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 0 if both corresponding bits of the first and second operands are 0. Otherwise, the corresponding bit in the result is set to 1.

8.46.1 Operation

Figure 61. OR Algorithm.

```
DEST ← SRC OR DEST;
EFLAGS.CF ← 0;
EFLAGS.OF ← 0;
EFLAGS.SF ← Sign(DEST);
EFLAGS.ZF ← Zero(DEST);
```

8.47 POP - Pop a Doubleword from the Stack

Opcode	Instruction
58	POP EAX
59	POP ECX
5A	POP EDX
5B	POP EBX
5C	POP ESP
5D	POP EBP
5E	POP ESI
5F	POP EDI
8F /0	POP r/m32
66 8F /0	POP r/m16

Loads the value from the top of the stack to the location specified by the operand (DEST) and then increments the stack pointer. The destination operand can be a general-purpose register or a memory location.

8.47.1 Operation

```
DEST←  [ESP];
IF Operand Size = 16 THEN
    /* Opcode 66 8F */
    ESP ←  ESP + 2;
ELSE
    ESP ←  ESP + 4;
ENDIF
```

8.48 POPFD - Pop Stack into EFLAGS Register

Opcode	Instruction
9D	POPFD

Pops a 32-bit value from the top of the stack and stores the value in the EFLAGS register. This instruction reverses the operation of the PUSHFD instruction.

8.48.1 Operation

Figure 62. Operation of POPFD

```
EFLAGS(CF,ZF,SF,IF,OF,TF) ←  [ESP];
ESP ←  ESP + 4;
```

8.49 PUSH - Push a Doubleword onto the Stack

Opcode	Instruction
50	PUSH EAX
51	PUSH ECX
52	PUSH EDX
53	PUSH EBX
54	PUSH ESP
55	PUSH EBP
56	PUSH ESI
57	PUSH EDI
68 id	PUSH imm32
66 68 id	PUSH imm16
6A ib	PUSH imm8

FF /6	PUSH r/m32
66 FF /6	PUSH r/m16

Decrements the stack pointer (ESP) by 4 then stores the operand (SRC) at the new address in ESP. The CPU sign extends 8-bit immediate values to 32-bits to preserve stack alignment. The CPU does not extend 16-bit immediate values and executing push imm16 (opcode 66h 68h) causes an unaligned stack.

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses an ESP relative address mode, the CPU computes the address of the operand before decrementing the ESP register.

```
IF Operand Size = 16 THEN
    /* Opcode 66 68 or 66 FF */
    ESP← ESP - 2;
    [ESP] ← SRC;
ELSE
    Temp ← SignExtend(SRC);
    ESP← ESP - 4;
    [ESP]← Temp;
ENDIF
```

8.50 PUSHFD - Push EFLAGS onto the Stack

Opcode	Instruction
9C	PUSHFD

Note: The PUSHFD instruction pushes the 32-bit EFLAGS register onto the stack.

8.50.1 Operation

```
ESP ← ESP - 4;
[ESP] ← EFLAGS;
```

8.51 RCL/RCR - Rotate Through Carry

Opcode	Instruction	Opcode	Instruction
C0 /2 ib	RCL r/m8, imm8	C0 /3 ib	RCR r/m8, imm8
66 C1 /2 ib	RCL r/m16, imm8	66 C1 /3 ib	RCR r/m16, imm8

C1 /2 ib	RCL r/m32, imm8	C1 /3 ib	RCR r/m32, imm8
D0 /2	RCL r/m8, 1	D0 /3	RCR r/m8, 1
66 D1 /2	RCL r/m16, 1	66 D1 /3	RCR r/m16, 1
D1 /2	RCL r/m32, 1	D1 /3	RCR r/m32, 1
D2 /2	RCL r/m8, CL	D2 /3	RCR r/m8, CL
66 D3 /2	RCL r/m16, CL	66 D3 /3	RCR r/m16, CL
D3 /2	RCL r/m32, CL	D3 /3	RCR r/m32, CL

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 CPU restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate through carry left (RCL) instruction shifts all bits toward more significant bit positions, except for the most-significant bit, which is rotated to the least significant bit. The Carry Flag is included in the rotation as if in bit position 33.

The rotate through carry right (RCR) instruction shifts all bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most significant bit. The Carry Flag is included in the rotation as if in bit position 33.

This instruction defines EFLAGS.OF only for 0 and 1 bit rotates. For rotates greater than 1 bit, EFLAGS.OF is undefined. For 0 bit rotates, flags are unaffected.

For 1 bit left rotates, the CPU sets the OF flag as the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For 1 bit right rotates, the CPU sets the OF flag to the exclusive OR of the two most-significant bits of the result.

8.52 RET - Return from Procedure

Opcode	Instruction
C3	RET

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

8.52.1 Operation

```
EIP ←  [ESP];
ESP ←  ESP + 4;
```

8.53 ROL/ROR - Rotate

Opcode	Instruction	Opcode	Instruction
C0 /0 ib	ROL r/m8, imm8	C0 /1 ib	ROR r/m8, imm8
66 C1 /0 ib	ROL r/m16, imm8	66 C1 /1 ib	ROR r/m16, imm8
C1 /0 ib	ROL r/m32, imm8	C1 /1 ib	ROR r/m32, imm8
D0 /0	ROL r/m8, 1	D0 /1	ROR r/m8, 1
66 D1 /0	ROL r/m16, 1	66 D1 /1	ROR r/m16, 1
D1 /0	ROL r/m32, 1	D1 /1	ROR r/m32, 1
D2 /0	ROL r/m8, CL	D2 /1	ROR r/m8, CL
66 D3 /0	ROL r/m16, CL	66 D3 /1	ROR r/m16, CL
D3 /0	ROL r/m32, CL	D3 /1	ROR r/m32, CL

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 CPU restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) instruction shifts all bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least significant bit.

The rotate right (ROR) instruction shifts all bits toward less-significant bit positions, except for the least-significant bit, which is rotated to the most significant bit.

For left rotates, the CPU sets the OF flag as the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the CPU sets the OF flag to the exclusive OR of the two most-significant bits of the result.

The EFLAGS.OF is defined only for 1-bit rotates. For rotates greater than 1 bit, the EFLAGS.OF is undefined. For left rotates, the CPU sets the OF flag as the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the CPU sets the OF flag to the exclusive OR of the two most-significant bits of the result.

8.54 SAL/SAR - Shift Arithmetic

Opcode	Instruction	Opcode	Instruction
C0 /4 ib	SAL r/m8, imm8	C0 /7 ib	SAR r/m8, imm8
66 C1 /4 ib	SAL r/m16, imm8	66 C1 /7 ib	SAR r/m16, imm8
C1 /4 ib	SAL r/m32, imm8	C1 /7 ib	SAR r/m32, imm8
D0 /4	SAL r/m8, 1	D0 /7	SAR r/m8, 1
66 D1 /4	SAL r/m16, 1	66 D1 /7	SAR r/m16, 1
D1 /4	SAL r/m32, 1	D1 /7	SAR r/m32, 1
D2 /4	SAL r/m8, CL	D2 /7	SAR r/m8, CL
66 D3 /4	SAL r/m16, CL	66 D3 /7	SAR r/m16, CL
D3 /4	SAL r/m32, CL	D3 /7	SAR r/m32, CL

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. The count range is limited to 0 to 31. A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) instruction shifts 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.

The shift arithmetic right (SAR) instruction shifts 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 set 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.

The SAR instruction can be used to perform signed division 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). EFLAGS.OF 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, the CPU sets EFLAGS.OF to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts.

8.55 SBB - Integer Subtraction with Borrow

Opcode	Instruction	Opcode	Instruction
18 /r	SBB r/m8, r8	66 1D iw	SBB AX, imm16
66 19 /r	SBB r/m16, r16	1D id	SBB EAX, imm32
19 /r	SBB r/m32, r32	80 /3 ib	SBB r/m8, imm8
1A /r	SBB r8, r/m8	66 81 /3 iw	SBB r/m16, imm16
66 1B /r	SBB r16, r/m16	81 /3 id	SBB r/m32, imm32
1B /r	SBB r32, r/m32	66 83 /3 ib	SBB r/m16, imm8
1C ib	SBB AL, imm8	83 /3 ib	SBB r/m32,imm8

Adds the second operand (SRC) and the carry (CF) flag, and subtracts the result from the first operand (DEST). The result of the subtraction is stored in the second operand (DEST). 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.

Before executing this instruction, the state of the CF flag represents a borrow from a previous subtraction. The subtraction operation treats EFLAGS.CF as an integer 1 or 0.

Immediate values are 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 EFLAGS.OF and EFLAGS.CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Software usually executes the SBB instruction as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

8.55.1 Operation

Figure 63. SBB Algorithm

```
Temp1 ← SignExtend(SRC);
Temp1 ← Temp1 + EFLAGS.CF;
DEST ← DEST - Temp1;
EFLAGS.CF ← Carry(DEST);
EFLAGS.ZF ← Zero(DEST);
EFLAGS.SF ← Sign(DEST);
EFLAGS.OF ← Overflow(DEST);
```

8.56 SHL/SHR - Shift

Opcode	Instruction	Opcode	Instruction
C0 /4 ib	SHL r/m8, imm8	C0 /5 ib	SHR r/m8, imm8
66 C1 /4 ib	SHL r/m16, imm8	66 C1 /5 ib	SHR r/m16, imm8
C1 /4 ib	SHL r/m32, imm8	C1 /5 ib	SHR r/m32, imm8
D0 /4	SHL r/m8, 1	D0 /5	SHR r/m8, 1
66 D1 /4	SHL r/m16, 1	66 D1 /5	SHR r/m16, 1
D1 /4	SHL r/m32, 1	D1 /5	SHR r/m32, 1
D2 /4	SHL r/m8, CL	D2 /5	SHR r/m8, CL
66 D3 /4	SHL r/m16, CL	66 D3 /5	SHR r/m16, CL
D3 /4	SHL r/m32, CL	D3 /5	SHR r/m32, CL

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. The count range is limited to 0 to 31. A special opcode encoding is provided for a count of 1.

The shift left (SHL) instruction shifts 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.

The shift right (SHR) instruction shifts 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 cleared.

Software may use the SHR instruction to perform unsigned division of the destination operand by powers of 2.

EFLAGS.OF 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, the CPU sets EFLAGS.OF to 1. For the SHR instruction, the OF flag is set to the most significant bit of the original operand.

8.57 SIDT - Store Interrupt Descriptor Table Register

Opcode	Instruction
0F 01 /1	SIDT m

8.57.1 Description

The SIDT instruction stores the Interrupt Descriptor Table Register (IDTR) to a 6 byte memory structure defined in "LIDT - Load Interrupt Descriptor Table Register" on page 82 for the LIDT instruction. The operand m is the memory address of the structure.

8.57.2 Exceptions

#UD	If the 66h prefix is used

8.58 STC - Set Carry Flag

Opcode	Instruction
F9	STC

Sets EFLAGS.CF. All other flags are unaffected.

8.58.1 Operation

Figure 64. STC Algorithm.

```
1 EFLAGS.CF ← 1;
```

8.59 STI - Set Interrupt Flag

Opcode	Instruction
FB	STI

Sets EFLAGS.IF to enable external maskable interrupts. If interrupts were disabled (ELFAGS.IF=0) when executing STI, the CPU may service an interrupt immediately after retiring this instruction.

Software must take special care when executing an STI immediately before a HLT instruction. In this case, the CPU may recognize an interrupt before executing the HLT. The interrupt service routine would then IRET to the HLT instruction which stops execution. If this behavior is not desired, the interrupt service routine can inspect the instruction pointed to by the EIP value on the interrupt stack frame. If the EIP points to a HLT (F4h) then the interrupt service routine can increment the value of the EIP in the stack frame. Incrementing the EIP causes the subsequent IRET to return to the next instruction after the HLT.

8.59.1 Operation

Figure 65. STI Algorithm.

```
1 EFLAGS.IF ← 1;
```

8.60 SUB - Subtract

Opcode	Instruction
28 /r	SUB r/m8, r8
66 29 /r	SUB r/m16, r16
29 /r	SUB r/m32, r32
2A /r	SUB r8, r/m8
66 2B /r	SUB r16, r/m16
2B /r	SUB r32, r/m32
2C ib	SUB AL, imm8
66 2D iw	SUB AX, imm16
2D id	SUB EAX, imm32
80 /5 ib	SUB r/m8, imm8
66 81 /5 iw	SUB r/m16, imm16
81 /5 id	SUB r/m32, imm32
66 83 /5 ib	SUB r/m16, imm8
83 /5 ib	SUB r/m32, imm8

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. The sign extends immediate operands to the length of the destination operand. The SUB instruction performs integer subtraction. The CPU 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.

8.60.1 Operation

Figure 66. SUB Algorithm

```
1 DEST ← DEST - SRC;
2 EFLAGS.CF ← Carry(DEST);
3 EFLAGS.ZF ← Zero(DEST);
4 EFLAGS.SF ← Sign(DEST);
5 EFLAGS.OF ← Overflow(DEST);
```

8.61 TEST - Logical Compare

Opcode	Instruction
84 /r	TEST r/m8, r8
66 85 /r	TEST r/m16, r16
85 /r	TEST r/m32, r32
A8 ib	TEST AL, imm8
66 A9 iw	TEST AX, imm16
A9 id	TEST EAX, imm32
F6 /0 ib	TEST r/m8, imm8
66 F7 /0 iw	TEST r/m16, imm16
F7 /0 id	TEST r/m32, imm32

8.61.1 Description

Performs a bitwise AND operation on the first operand (SRC1) and second operand (SRC2) and sets the SF and ZF status flags according to the result. The result is then discarded.

8.61.2 Operation

Figure 67. TEST Algorithm

```
Temp ← SRC1 AND SRC2;
EFLAGS.SF ← Sign(Temp);
IF Temp = 0 THEN
    EFLAGS.ZF ← 1;
ELSE
    EFLAGS.ZF ← 0;
ENDIF
EFLAGS.CF ← 0;
EFLAGS.OF ← 0;
```

8.62 UD2 - Undefined Instruction

Opcode	Instruction
0F 0B	UD2

Generates an Invalid Opcode Fault (#UD). This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcode for this instruction is reserved for this purpose. Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

The instruction pointer in the exception stack frame references the UD2 instruction and not the following instruction.

Note: Some disassemblers such as the GNU objdump utility use UD2A for this opcode.

8.62.1 Exceptions

#UD This instruction always causes this exception.

8.63 XOR - Logical Exclusive OR

Opcode	Instruction
30 /r	XOR r/m8, r8
66 31 /r	XOR r/m16, r16
31 /r	XOR r/m32, r32
32 /r	XOR r8, r/m8
66 33 /r	XOR r16, r/m16
33 /r	XOR r32, r/m32
34 ib	XOR AL, imm8
66 35 iw	XOR AX, imm16
35 id	XOR EAX, imm32
80 /6 ib	XOR r/m8, imm8
66 81 /6 iw	XOR r/m16, imm16
81 /6 id	XOR r/m32, imm32
83 /6 ib	XOR r/m16, imm8
83 /6 ib	XOR r/m32, imm8

Performs a bitwise exclusive-OR (XOR) operation on the first operand (DEST) and the second operand (SRC) and stores the result in the first operand. The source operand can be an immediate, a register, or a memory location. The destination operand can be a register or a memory location. 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.

8.63.1 Operation

Figure 68. XOR Algorithm.

```
1 DEST ← SRC XOR DEST;
2 EFLAGS.CF ← 0;
3 EFLAGS.ZF ← Zero(DEST);
4 EFLAGS.SF ← Sign(DEST);
5 EFLAGS.OF ← 0;
```

§ §

Appendix A Porting From IA

The following sections show software substitutions for some unsupported IA-32 instructions. The following sections also list key functional differences between IA-32 and Intel® Quark™ microcontroller D1000 CPU instructions.

A.1 PUSHA

The following code emulates the action of the PUSHA instruction:

```
# pusha emulation start
pushl   %eax
pushl   %ecx
pushl   %edx
pushl   %ebx
pushl   %esp            # pushes esp value before decrement
addl    $16, (%esp)
pushl   %ebp
pushl   %esi
pushl   %edi
# pusha emulation end
```

A.2 POPA

The following code emulates the action of the POPA instruction:

```
# popa emulation start
popl    %edi
popl    %esi
popl    %ebp
popl    %ebx    #dummy pop
popl    %ebx
popl    %edx
popl    ecx
popl    %eax
# popoa emulation end
```

A.3 XCHG

The CPU does not support the XCHG instruction, except for the special NOP cases:

```
xchg    %eax, %eax  # one byte NOP
xchg    %ax, %ax    # one byte NOP with 66h prefix
```

In general use, you can replace XCHG with a three instruction sequence as shown in the following example:

```
# xchg      %eax,%ebx
# xchg emulation start
pushl   %ebx
movl    %eax,%ebx
popl    %eax
# xchg emulation end
```

If your code can ignore the resulting EFLAGS value, then replace XCHG with 3 xor operations. This substitution executes faster by eliminating the push/pop memory touches.

```
# xchg      %eax,%ebx
# xchg emulation start
xor     %eax,%ebx
xor     %ebx,%eax
xor     %eax,%ebx
# xchg emulation end
```

Exchanging a register with the stack pointer (ESP) requires special handling as shown:

```
# xchg      %esp,%ebp
# xchg emulation start
pushl   %ebp            # save ebp, esp too low by 4
lea     4(%esp),%ebp    # copy low esp to ebp and fixup ebp
popl    %esp            # copy ebp to esp
# xchg emulation end
```

Note: The XCHG instruction also implies a locked transaction. These substitutions do not provide any locking.

A.4 Instruction Prefixes

The CPU supports a subset of the IA-32 instruction prefix possibilities. See Table 24.

Note: Only the Pentium 4 implemented branch hint prefixes. All other IA processors ignore branch hints.

Table 24. Instruction Prefix Bytes

Prefix Byte (hex)	Description	Supported?
66	16-bit Operand Size	Yes
67	16-bit Address Size	No
F0	Lock	Yes
F3	REP/REPE/REPZ	No
F2	REPNE/REPNZ	No
2E,36,3E,26,64,65	Segment Overrides	No
2E,3E	Branch Hints	No

Note: The CPU does not support IA-32 prefixes shaded gray.

A.5 INT and INT3

The CPU INT instruction with operand value of 3 behaves identically to the INT3 instruction.

A.6 Interrupt Descriptors

Intel® Quark™ microcontroller D1000 processor supports a subset of IA-32 Interrupt Descriptor functionality. Intel® Quark™ microcontroller D1000 processor supports only descriptors for a 32-bit address space and only the Interrupt Gate and Trap Gate types. Furthermore, many fields in the IA-32 descriptor format are reserved in Intel® Quark™ microcontroller D1000 processor.

Note: The description of the IA-32 IDT is in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 Chapter 6.

A.7 IO Instructions

The CPU does not implement the IA-32 IN and OUT instructions. Accordingly, the CPU does not have the concept of IO Privilege Level (IOPL) and does not implement the EFLAGS IOPL bits.

Developers must substitute memory mapped IO (MMIO) accesses in place of IN and OUT instructions. Designers should map devices with MMIO interfaces in the Intel® Quark™ microcontroller D1000 processor strongly ordered memory region. For information on strongly ordered memory, see Section 3.3.

A.8 EFLAGS

Intel® Quark™ microcontroller D1000 processor supports a status register called EFLAGS that resembles the IA- 32 EFLAGS register. The behavior of the arithmetic EFLAGS bits, namely CF, OF, SF and ZF is upward compatible with IA-32 processors. Table 24 highlights differences from IA-32. Notable exceptions are the EFLAGS.L1,0 bits which are unique to Intel® Quark™ microcontroller D1000 processor.

Figure 69. Flags Defined in the EFLAGS Register

31–12	11	10	9	8	7	6	5–2	1	0
0	O	0	I	T	S	Z	0	1	C

Flag	Bit	Type	Supported?	Description
CF	0	Status	Yes	Carry Flag
	1	Fixed	Yes	Reserved
	2	Fixed	No	Reserved
	3	Fixed	No	Reserved
	4	Fixed	No	Reserved
	5	Fixed	No	Reserved
ZF	6	Status	Yes	Zero Flag
SF	7	Status	Yes	Sign Flag
TF	8	System	Yes	Trap Flag
IF	9	System	Yes	Interrupt Enable Flag
	10	Fixed	No	Reserved
OF	11	Status	Yes	Overflow Flag
	12-31	Fixed	Yes	AI-32 Reserved

Note: Intel® Quark™ microcontroller D1000 processor does not support IA- 32 EFLAGS bits shaded gray.

A.9 Exceptions

Intel® Quark™ microcontroller D1000 processor supports an exception vector table which is a generally a subset of IA-32. Table 25 highlights the differences.

Table 25. Interrupt Descriptor Table (IDT)

Vector	Name	Type	Error Code?	Description
0	#DE	Fault	No	Divide by 0
1	#DB	Trap	No	Debug Exception
2			No	Reserved
3	#BP	Trap	No	Breakpoint (INT3)
4	#OF	Trap	No	Reserved
5	#BR	Fault	No	Reserved
6	#UD	Fault	No	Invalid Opcode
7	#NM	Fault	No	Reserved
8	#DF	Abort	Yes	Double Fault
9		Fault	No	Reserved
10	#TS	Fault	No	Reserved

Table 25. Interrupt Descriptor Table (IDT)

Vector	Name	Type	Error Code?	Description
11	#NP	Fault	Yes	Not Present
12	#SS	Fault	No	Reserved
13	#GP	Fault	Yes	General Protection
14	#PF	Fault	No	Reserved
15				Intel Reserved
16	#MF	Fault	No	Reserved
17	#AC	Fault	No	Reserved
18	#MC	Fault	Yes	Machine Check (non-IA)
19	#XM	Fault	No	Reserved
20-31				Intel Reserved
32-255		Interrupt	No	Asynchronous IRQ

Note: Shaded areas denote IA-32 exceptions not supported by Intel® Quark™ microcontroller D1000 processor.

A.10 Segmentation

Intel® Quark™ microcontroller D1000 processor does not support any form of segmentation. All addresses are linear as described in Chapter 3.0. Modern IA-32 code configures segmentation to behave in a pass-through manner and in most cases porting code to Intel® Quark™ microcontroller D1000 processor requires no effort in this regard. The notable exception is IA-32 threadlocal storage (TLS). Quark D1000 does not currently support TLS and TLS specific code must be reimplemented for Intel® Quark™ microcontroller D1000 processor.

§ §

Appendix B IOAPIC Programming Examples

This section provides IOAPIC Programming examples.

B.1 Masking Interrupts

These functions enable or disable the specified IOAPIC interrupt input while preserving the remaining bits of the Redirection Entry Register associated with the interrupt.

```
#define IOREGSEL ((volatile unsigned int *)0xFEC00000)
#define IOWIN ((volatile unsigned int *)0xFEC00010)
#define MASK_BIT 0x00010000

/*
 * Disable external interrupt.
 * Offset is 0x10 plus 8 bytes per irq.
 *
 * irq: The interrupt input number on the IOAPIC
 *      Range from 0 to N
 */
void mask_irq( unsigned int irq )
{
        *IOREGSEL = 0x10 + irq * 2;
        *IOWIN |= MASK_BIT;
}

/*
 * Enable external interrupt
 * Offset is 0x10 plus 8 bytes per irq.
 *
 * irq: The interrupt input number on the IOAPIC
 *      Range from 0 to N
 */
void unmask_irq( unsigned int irq )
{
        *IOREGSEL = 0x10 + irq * 2;
        *IOWIN &= ~MASK_BIT;
}
```

§ §