Intel® Arc™ A-Series Graphics and Intel Data Center GPU Flex Series

Open-Source Programmer's Reference Manual

For the discrete GPUs code named "Alchemist" and "Arctic Sound-M"

Volume 9: Render Engine

March 2023, Revision 1.0

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks

Customer is responsible for safety of the overall system, including compliance with applicable safety-related requirements or standards.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exceptions that a) you may publish an unmodified copy and b) code included in this document is licensed subject to Zero-Clause BSD open source license (0BSD). You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Table of Contents

Render Engine

The Render Engine supports command streams used both for 3D and Compute (GPGPU) workloads. These command streams fetch the data, and dispatch individual work items to many threads that operate in parallel. The threads run small software programs (also called kernels or shaders) on the GPU processors (called Execution Units).

The command streamers control the programmable pipelines in the Render Engine so that the individual programs run in parallel but are synchronized to start only when their required data is available, and complete when all the work is done.

Each pipeline in the Render Engine shares common state with all the threads running in the pipeline. The command streamer manages that state.

Workload Submission

This section describes workload submission on the graphics engine.

Multi-context Submission Overview

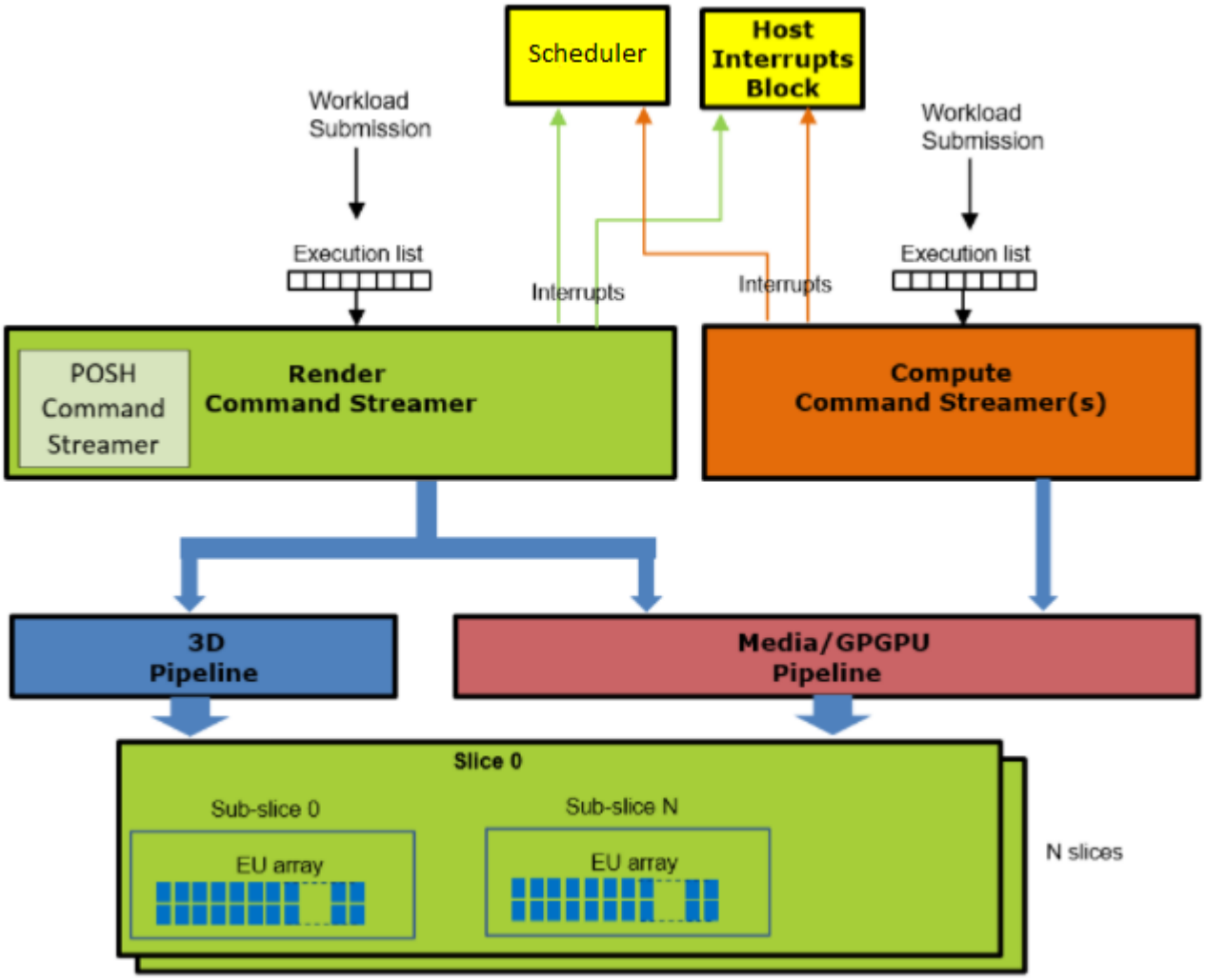
Work into the Graphics engine is input using the Command Streamer or multiple command streamers.

Each command streamers are independent and support their own submit port and execution list. Within a command streamer, the work elements are executed serially.

The Render Command Streamer runs in one of the following modes (that is specified using the PIPELINE_SELECT command):

- 3D
- Media/GPGPU

The Compute Command Streamer only supports the Media/GPGPU mode. Number of Compute CS varies per SKU.


Scheduler
Host Interrupts Block
Workload Submission
Workload Submission
Execution list
Execution list
Interrupts
Interrupts
POSH Command Streamer
Render Command Streamer
Compute Command Streamer(s)
3D Pipeline
Media/GPGPU Pipeline
Slice 0
Sub-slice 0
Sub-slice N
EU array
EU array
N slices

Render-3D-GPGPU Command Streamer

This section describes the infrastructure provided by the Command Streamer of the Render engine which supports 3D, Compute and Programmable Media.

Batch Buffer Privilege Register

FORCE_TO_NONPRIV - FORCE_TO_NONPRIV

Context Save Registers

The following are the Context Save Registers:

Register
BB_PREEMPT_ADDR - Batch Buffer Head Pointer Preemption Register
BB_PREEMPT_ADDR_UDW - Batch Buffer Upper Head Pointer Preemption Register
RING_BUFFER_HEAD_PREEMPT_REG - RING_BUFFER_HEAD_PREEMPT_REG
BB_START_ADDR - Batch Buffer Start Head Pointer Register
BB_START_ADDR_UDW - Batch Buffer Start Upper Head Pointer Register
BB_ADDR_DIFF - Batch Address Difference Register
BB_OFFSET - Batch Offset Register
SBB_PREEMPT_ADDR - Second Level Batch Buffer Head Pointer Preemption Register
SBB_PREEMPT_ADDR_UDW - Second Level Batch Buffer Upper Head Pointer Preemption Register
BB_PER_CTX_PTR - Batch Buffer Per Context Pointer

Mode Registers

The following are the Mode Registers:

Register
INSTPM - Instruction Parser Mode Register
EXCC - Execute Condition Code Register
NOPID - NOP Identification Register
CSPREEMPT - CSPREEMPT
IDLEDLY - Idle Switch Delay
SEMA_WAIT_POLL - Semaphore Polling Interval on Wait
HWS_PGA - Hardware Status Page Address Register

Watchdog Timer Registers

These registers together implement a watchdog timer. Writing ones to the control register enables the counter, and writing zeros disables the counter. The second register is programmed with a threshold value which, when reached, signals an interrupt that then resets the counter to 0. Program the threshold value before enabling the counter or extremely frequent interrupts may result.

Note: The counter itself is not observable. It increments with the main render clock.

Programming Notes: When watch dog timer is enabled, HW does not trigger any kind of idle sequences. SW must enable and disable watch dog timer for any given workload within the same command buffer dispatch. SW must disable watch dog timer around semaphore waits and wait for events commands so that HW can trigger appropriate idle sequence for power savings.

Logical Context Support

The following are the Logical Context Support Registers:

Register
BB_ADDR - Batch Buffer Head Pointer Register
BB_ADDR_UDW - Batch Buffer Upper Head Pointer Register
CXT_SIZE - Context Sizes
CXT_EL_OFFSET - Exec-List Context Offset
SYNC_FLIP_STATUS - Wait For Event and Display Flip Flags Register
SYNC_FLIP_STATUS_1 - Wait For Event and Display Flip Flags Register 1
SYNC_FLIP_STATUS_2 - Wait For Event and Display Flip Flags Register 2
WAIT_FOR_RC6_EXIT - Control Register for Power Management
SBB_ADDR - Second Level Batch Buffer Head Pointer Register
SBB_ADDR_UDW - Second Level Batch Buffer Upper Head Pointer Register
SBB_STATE - Second Level Batch Buffer State Register
PS_INVOCATION_COUNT_SLICE0 - PS Invocation Count for Slice0
PS_INVOCATION_COUNT_SLICE1 - PS Invocation Count for Slice1
PS_INVOCATION_COUNT_SLICE2 - PS Invocation Count for Slice2
PS_DEPTH_COUNT_SLICE0 - PS Depth Count for Slice0
PS_DEPTH_COUNT_SLICE1 - PS Depth Count for Slice1
PS_DEPTH_COUNT_SLICE2 - PS Depth Count for Slice2
R_PWR_CLK_STATE - Render Power Clock State Register

MI Commands for Render Engine

This chapter describes the formats of the "Memory Interface" commands, including brief descriptions of their use. The functions performed by these commands are discussed fully in the *Memory Interface Functions* Device Programming Environment chapter.

This chapter describes MI Commands for the original graphics processing engine. The term "for Rendering Engine" in the title has been added to differentiate this chapter from a similar one describing the MI commands for the Media Decode Engine.

The commands detailed in this chapter are used across product families. However, slight changes may be present in some commands (i.e., for features added or removed), or some commands may be removed entirely. Refer to the *Preface* chapter for product specific summary.

Commands
MI_NOOP
MI_ARB_CHECK
MI_ARB_ON_OFF
MI_BATCH_BUFFER_START
MI_BATCH_BUFFER_END
MI_CONDITIONAL_BATCH_BUFFER_END
MI_DISPLAY_FLIP
MI_LOAD_SCAN_LINES_EXCL
MI_LOAD_SCAN_LINES_INCL
MI_REPORT_HEAD
MI_STORE_DATA_IMM
MI_ATOMIC
MI_COPY_MEM_MEM
MI_LOAD_REGISTER_REG
MI_LOAD_REGISTER_MEM
MI_STORE_REGISTER_MEM
MI_SUSPEND_FLUSH
MI_USER_INTERRUPT
MI_WAIT_FOR_EVENT
MI_SEMAPHORE_SIGNAL
MI_SEMAPHORE_WAIT
MI_FORCE_WAKEUP

Render Engine Command Streamer

The RCS (Render Command Streamer) unit primarily serves as the software programming interface between the O/S driver and the Render Engine. It is responsible for fetching, decoding, and dispatching of data packets (3D/Media Commands with the header DWord removed) to the front-end interface module of Render Engine.

Logic Functions Included
• MMIO register programming interface. • DMA action for fetching of ring data from memory. • Management of the Head pointer for the Ring Buffer. • Decode of ring data and sending it to the appropriate destination: 3D (Vertex Fetch Unit) & GPGPU. • Handling of user interrupts. • Flushing the 3D and GPGPU Engine. • Handle NOP.
• DMA action for fetching of execlists from memory. • Handling of ring context switch interrupt.

The RCS unit only claims memory mapped I/O cycles that are targeted to its range of 0x2000 to 0x27FF. The Gx and MFX Engines use semaphore to synchronize their operations.

RCS operates completely independent of the MFx CS.

The simple sequence of events is as follows: a ring (say PRB0) is programmed by a memory-mapped register write cycle. The DMA inside RCS is kicked off. The DMA fetches commands from memory based on the starting address and head pointer. The DMA requests cache lines from memory (one cacheline CL at a time). There is guaranteed space in the DMA FIFO (8 CL deep) for data coming back from memory. The DMA control logic has copies of the head pointer and the tail pointer. The DMA increments the head pointer after making requests for ring commands. Once the DMA copy of the head pointer becomes equal to the tail pointer, the DMA stops requesting.

The parser starts executing once the DMA FIFO has valid commands. All the commands have a header DWord packet. Based on the encoding in the header packet, the command may be targeted towards Vertex Fetch Unit or GPPGU engine or the command parser. After execution of every command, the actual head pointer is updated. The ring is considered empty when the head pointer becomes equal to the tail pointer.

Render Command Formats

3D Commands

The 3D commands are used to program the graphics pipelines for 3D operations.

Refer to the *3D* chapter for a description of the 3D state and primitive commands and the *Media* chapter for a description of the media-related state and object commands.

For all commands listed in **3D Command Map**, the Pipeline Type (bits 28:27) is 3h, indicating the 3D Pipeline.

3D Command Map

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	01h	Reserved	3D Pipeline
0h	02h	Reserved	3D Pipeline
0h	03h	Reserved	
0h	04h	3DSTATE_CLEAR_PARAMS	3D Pipeline
0h	05h	3DSTATE_DEPTH_BUFFER	3D Pipeline
0h	06h	3DSTATE_STENCIL_BUFFER	3D Pipeline
0h	07h	3DSTATE_HIER_DEPTH_BUFFER	3D Pipeline
0h	08h	3DSTATE_VERTEX_BUFFERS	Vertex Fetch
0h	09h	3DSTATE_VERTEX_ELEMENTS	Vertex Fetch
0h	0Ah	3DSTATE_INDEX_BUFFER	Vertex Fetch
0h	0Bh	3DSTATE_VF_STATISTICS	Vertex Fetch
0h	0Ch	3DSTATE_VF	Vertex Fetch
0h	0Dh	3DSTATE_VIEWPORT_STATE_POINTERS	3D Pipeline
0h	0Eh	3DSTATE_CC_STATE_POINTERS	3D Pipeline
0h	10h	3DSTATE_VS	Vertex Shader
0h	11h	3DSTATE_GS	Geometry Shader
0h	12h	3DSTATE_CLIP	Clipper
0h	13h	3DSTATE_SF	Strips & Fans
0h	14h	3DSTATE_WM	Windower
0h	15h	3DSTATE_CONSTANT_VS	Vertex Shader
0h	16h	3DSTATE_CONSTANT_GS	Geometry Shader
0h	17h	3DSTATE_CONSTANT_PS	Windower
0h	18h	3DSTATE_SAMPLE_MASK	Windower
0h	19h	3DSTATE_CONSTANT_HS	Hull Shader
0h	1Ah	3DSTATE_CONSTANT_DS	Domain Shader
0h	1Bh	3DSTATE_HS	Hull Shader
0h	1Ch	3DSTATE_TE	Tesselator
0h	1Dh	3DSTATE_DS	Domain Shader
0h	1Eh	3DSTATE_STREAMOUT	HW Streamout
0h	1Fh	3DSTATE_SBE	Setup
0h	20h	3DSTATE_PS	Pixel Shader
0h	21h	3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP	Strips & Fans
0h	22h	3DSTATE_CPS_POINTER	Course Pixel Shader
0h	23h	3DSTATE_VIEWPORT_STATE_POINTERS_CC	Windower
0h	24h	3DSTATE_BLEND_STATE_POINTERS	Pixel Shader

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	25h	3DSTATE_DEPTH_STENCIL_STATE_POINTERS	Pixel Shader
0h	26h	3DSTATE_BINDING_TABLE_POINTERS_VS	Vertex Shader
0h	27h	3DSTATE_BINDING_TABLE_POINTERS_HS	Hull Shader
0h	28h	3DSTATE_BINDING_TABLE_POINTERS_DS	Domain Shader
0h	29h	3DSTATE_BINDING_TABLE_POINTERS_GS	Geometry Shader
0h	2Ah	3DSTATE_BINDING_TABLE_POINTERS_PS	Pixel Shader
0h	2Bh	3DSTATE_SAMPLER_STATE_POINTERS_VS	Vertex Shader
0h	2Ch	3DSTATE_SAMPLER_STATE_POINTERS_HS	Hull Shader
0h	2Dh	3DSTATE_SAMPLER_STATE_POINTERS_DS	Domain Shader
0h	2Eh	3DSTATE_SAMPLER_STATE_POINTERS_GS	Geometry Shader
0h	2Fh	3DSTATE_SAMPLER_STATE_POINTERS_PS	Pixel Shader
0h	30h	3DSTATE_URB_VS	Vertex Shader
0h	31h	3DSTATE_URB_HS	Hull Shader
0h	32h	3DSTATE_URB_DS	Domain Shader
0h	33h	3DSTATE_URB_GS	Geometry Shader
0h	34h	3DSTATE_GATHER_CONSTANT_VS	Vertex Shader
0h	35h	3DSTATE_GATHER_CONSTANT_GS	Geometry Shader
0h	36h	3DSTATE_GATHER_CONSTANT_HS	Hull Shader
0h	37h	3DSTATE_GATHER_CONSTANT_DS	Domain Shader
0h	38h	3DSTATE_GATHER_CONSTANT_PS	Pixel Shader
0h	39h	3DSTATE_DX9_CONSTANTF_VS	Vertex Shader
0h	3Ah	3DSTATE_DX9_CONSTANTF_PS	Pixel Shader
0h	3Bh	3DSTATE_DX9_CONSTANTI_VS	Vertex Shader
0h	3Ch	3DSTATE_DX9_CONSTANTI_PS	Pixel Shader
0h	3Dh	3DSTATE_DX9_CONSTANTB_VS	Vertex Shader
0h	3Eh	3DSTATE_DX9_CONSTANTB_PS	Pixel Shader
0h	3Fh	3DSTATE_DX9_LOCAL_VALID_VS	Vertex Shader
0h	40h	3DSTATE_DX9_LOCAL_VALID_PS	Pixel Shader
0h	41h	3DSTATE_DX9_GENERATE_ACTIVE_VS	Vertex Shader
0h	42h	3DSTATE_DX9_GENERATE_ACTIVE_PS	Pixel Shader
0h	43h	3DSTATE_BINDING_TABLE_EDIT_VS	Vertex Shader
0h	44h	3DSTATE_BINDING_TABLE_EDIT_GS	Geometry Shader
0h	45h	3DSTATE_BINDING_TABLE_EDIT_HS	Hull Shader
0h	46h	3DSTATE_BINDING_TABLE_EDIT_DS	Domain Shader
0h	47h	3DSTATE_BINDING_TABLE_EDIT_PS	Pixel Shader
0h	48h	3DSTATE_VF_HASHING	Vertex Fetch

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	49h	3DSTATE_VF_INSTANCING	Vertex Fetch
0h	4Ah	3DSTATE_VF_SGVS	Vertex Fetch
0h	4Bh	3DSTATE_VF_TOPOLOGY	Vertex Fetch
0h	4Ch	3DSTATE_WM_CHROMA_KEY	Windower
0h	4Dh	3DSTATE_PS_BLEND	Windower
0h	4Eh	3DSTATE_WM_DEPTH_STENCIL	Windower
0h	4Fh	3DSTATE_PS_EXTRA	Windower
0h	50h	3DSTATE_RASTER	Strips & Fans
0h	51h	3DSTATE_SBE_SWIZ	Strips & Fans
0h	52h	3DSTATE_WM_HZ_OP	Windower
0h	53h	3DSTATE_INT (internally generated state)	3D Pipeline
0h	54h	3DSTATE_RS_CONSTANT_POINTER	Resource Streamer
0h	55h	3DSTATE_VF_COMPONENT_PACKING	Vertex Fetch
0h	56h	3DSTATE_VF_SGVS_2	VertexFetch
0h	57h	3DSTATE_VFG	VertexFetch
0h	58h	3DSTATE_URB_ALLOC_VS	VertexShader
0h	59h	3DSTATE_URB_ALLOC_HS	HullShader
0h	5Ah	3DSTATE_URB_ALLOC_DS	DomainShader
0h	5Bh	3DSTATE_URB_ALLOC_GS	GeometryShader
0h	5Dh-5Fh	Reserved	
0h	60h	3DSTATE_SO_BUFFER_INDEX_0	HW StreamOut
0h	61h	3DSTATE_SO_BUFFER_INDEX_1	HW StreamOut
0h	62h	3DSTATE_SO_BUFFER_INDEX_2	HW StreamOut
0h	63h	3DSTATE_SO_BUFFER_INDEX_3	HW StreamOut
0h	64h-69h	Reserved	
0h	6Ah	3DSTATE_PTBR_MARKER	3D Pipeline
0h	6Bh	3DSTATE_PTBR_TILE_SELECT	Vertex Fetch, Strips & Fans
0h	6Ch	3DSTATE_PRIMITIVE_REPLICATION	3D Pipeline
0h	6Dh	3DSTATE_CONSTANT_ALL	3D Pipeline
0h	6Eh	3DSTATE_TBIMR_TILE_PASS_INFO	SF
0h	6Fh	3DSTATE_AMFS	PSS
0h	70h	3DSTATE_DEPTH_CNTL_BUFFER	WM
0h	71h	3DSTATE_DEPTH_BOUNDS	WM
0h	72h	3DSTATE_AMFS_TEXTURE_POINTERS	WM
0h	73h	3DSTATE_CONSTANT_TS_POINTER	PSS
0h	77h	3DSTATE_MESH_CONTROL	GS

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	78h	3DSTATE_MESH_DISTRIB	VFG
0h	79h	3DSTATE_TASK_REDISTRIB	HS
0h	7Ah	3DSTATE_MESH_SHADER	TE
0h	7Bh	3DSTATE_MESH_SHADER_DATA	GS
0h	7Ch	3DSTATE_TASK_CONTROL	HS
0h	7Dh	3DSTATE_TASK_SHADER	HS
0h	7Eh	3DSTATE_TASK_SHADER_DATA	HS
0h	7Fh	3DSTATE_URB_ALLOC_MESH	GS
0h	80h	3DSTATE_URB_ALLOC_TASK	HS
0h	81h	3DSTATE_CLIP_MESH	CL
0h	82h	3DSTATE_SBE_MESH	SBE
0h	57h-59h	Reserved	
0h	60h-68h	Reserved	
0h	69h	Reserved	
0h	74h	Reserved	
0h	75h	Reserved	
0h	76h	Reserved	
0h	77h-82h	Reserved	
0h	83h	3DSTATE_CPSIZE_CONTROL_BUFFER	WM
0h	83h-FFh	Reserved	
1h	00h	3DSTATE_DRAWING_RECTANGLE	Strips & Fans
1h	02h	Reserved	
1h	03h	Reserved	
1h	04h	3DSTATE_CHROMA_KEY	Sampling Engine
1h	05h	Reserved	
1h	06h	3DSTATE_POLY_STIPPLE_OFFSET	Windower
1h	07h	3DSTATE_POLY_STIPPLE_PATTERN	Windower
1h	08h	3DSTATE_LINE_STIPPLE	Windower
1h	0Ah	3DSTATE_AA_LINE_PARAMS	Windower
1h	0Bh	3DSTATE_GS_SVB_INDEX	Geometry Shader
1h	0Ch	Reserved	
1h	0Dh	3DSTATE_MULTISAMPLE	Windower
1h	0Eh	3DSTATE_STENCIL_BUFFER	Windower
1h	0Fh	3DSTATE_HIER_DEPTH_BUFFER	Windower
1h	10h	3DSTATE_CLEAR_PARAMS	Windower
1h	11h	3DSTATE_MONOFILTER_SIZE	Sampling Engine

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
1h	12h	3DSTATE_PUSH_CONSTANT_ALLOC_VS	Vertex Shader
1h	13h	3DSTATE_PUSH_CONSTANT_ALLOC_HS	Hull Shader
1h	14h	3DSTATE_PUSH_CONSTANT_ALLOC_DS	Domain Shader
1h	15h	3DSTATE_PUSH_CONSTANT_ALLOC_GS	Geometry Shader
1h	16h	Reserved	
1h	17h	3DSTATE_SO_DECL_LIST	HW Streamout
1h	18h	3DSTATE_SO_BUFFER	HW Streamout
1h	19h	3DSTATE_BINDING_TABLE_POOL_ALLOC	Resource Streamer
1h	1Ah	3DSTATE_GATHER_POOL_ALLOC	Resource Streamer
1h	1Bh	3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC	Resource Streamer
1h	1Ch	3DSTATE_SAMPLE_PATTERN	Windower
1h	1Dh	3DSTATE_URB_CLEAR	3D Pipeline
1h	1Eh	3DSTATE_3D_MODE	3D Pipeline
1h	1Fh	3DSTATE_SUBSLICE_HASH_TABLE	3D Pipeline
1h	20h	3DSTATE_SLICE_TABLE_STATE_POINTERS	3D Pipeline
1h	21h	3DSTATE_PTBR_PAGE_POOL_BASE_ADDRESS	3D Pipeline
1h	22h	3DSTATE_PTBR_TILE_PASS_INFO	3D Pipeline
1h	23h	3DSTATE_PTBR_RENDER_LIST_BASE_ADDRESS	3D Pipeline
1h	24h	3DSTATE_PTBR_FREE_LIST_BASE_ADDRES	3D Pipeline
1h	23h-2Ah	Reserved	
1h	2Bh-FFh	Reserved	
2h	00h	PIPE_CONTROL	Render/Compute Pipeline
2h	01h	Reserved	
2h	03h-FFh	Reserved	
3h	00h	3DPRIMITIVE	Vertex Fetch
3h	01h	3DMESH_1D	Vertex Fetch
3h	02h	3DMESH_3D	Vertex Fetch
3h	01h	Reserved	
3h	02h	Reserved	
3h	03h-FFh	Reserved	
4h	00h-01h	Reserved	
5h-7h	00h-FFh	Reserved	

Pipeline Type (28:27)	Opcode	Sub Opcode	Command	Definition Chapter
Common (pipelined)	Bits 26:24	Bits 23:16		
0h	0h	04h-FFh	Reserved	
Common (non-pipelined)	Bits 26:24	Bits 23:16		
0h	1h	00h	Reserved	N/A
0h	1h	01h	STATE_BASE_ADDRESS	Graphics Processing Engine
0h	1h	02h	STATE_SIP	Graphics Processing Engine
0h	1h	03h	Reserved	3D Pipeline
0h	1h	04h	GPGPU CSR BASE ADDRESS	Graphics Processing Engine
0h	1h	05h	STATE_COMPUTE_MODE	Compute Pipeline
0h	1h	06h	3DSTATE_BTD	
0h	1h	07h-08h	Reserved	
0h	1h	09h	Reserved	
0h	1h	0Ah-FFh	Reserved	N/A
Reserved	Bits 26:24	Bits 23:16		
0h	2h-7h	XX	Reserved	N/A

Render Command Header Format

Render Command Header

Type	Bits				
	31:29	28:24	23	22	21:0
Memory Interface (MI)	000	Opcode 00h - NOP 0Xh - Single DWord Commands 1Xh - Two+ DWord Commands 2Xh - Store Data Commands 3Xh - Ring/Batch Buffer Cmds		Identification No./DWord Count Command Dependent Data 5:0 - DWord Count 5:0 - DWord Count 5:0 - DWord Count	

Type	Bits					
	31:29	28:24	23	22:17	16:10	9:0

Type	Bits				
	31:29	28:24	23:19	18:16	15:0
Reserved	001, 010	Opcode - 11111	Sub Opcode 00h - 01h	Reserved	DWord Count

Type	Bits					
	31:29	28:27	26:24	23:16	15:8	7:0
Common	011	00	Opcode - 000	Sub Opcode	Data	DWord Count
Common (NP)[1]	011	00	Opcode - 001	Sub Opcode	Data	DWord Count
Reserved	011	00	Opcode - 010 - 111			
Single Dword Command	011	01	Opcode - 000 - 001	Sub Opcode		N/A
Reserved	011	01	Opcode - 010 - 111			
Media State	011	10	Opcode - 000	Sub Opcode		Dword Count
Media Object	011	10	Opcode - 001 - 010	Sub Opcode	Dword Count	
Reserved	011	10	Opcode - 011 - 111			
3DState (Pipelined)	011	11	Opcode - 000	Sub Opcode	Data	DWord Count
3DState (NP)[1]	011	11	Opcode - 001	Sub Opcode	Data	DWord Count
PIPE_Control	011	11	Opcode - 010		Data	DWord Count
3DPrimitive	011	11	Opcode - 011		Data	DWord Count
Reserved	011	11	Opcode - 100			
L3_CONTROL	011	11	Opcode-101			
Reserved	011	11	Opcode - 110 - 111			
Reserved	100	XX				
Reserved	101	XX				
Reserved	110	XX				

Notes:

[1]The qualifier "NP" indicates that the state variable is non-pipelined and the render pipe is flushed before such a state variable is updated. The other state variables are pipelined (default).

Render Engine Preemption

Render Engine Command Streamer Preemptable Commands

Preemptable Command	Condition
MI_ARB_CHECK	AP
Element Boundary	AP (if allowed)
Semaphore Wait	Unsuccessful & AP
Wait for Event	Unsuccessful & AP (if allowed)
3DPRIMITIVE	Object Level (if enabled[1])
GPGPU_WALKER	Mid-Thread (if enabled[2])
PIPE_CONTROL[3]	PIPESEL_GPGPU MODE / PIPESEL-MEDIA MODE
MEDIA STATE FLUSH	Mid-Thread (if enabled[2])
MEDIA_OBJECT_WALKER / MEDIA_OBJECT	Thread Group

Preemptable Command	Condition
PIPELINE_SELECT	PIPESEL-GPGPU Mode / PIPESEL-MEDIA MODE
Any Non-Pipelined State[4]	PIPESEL-GPGPU Mode / PIPESEL-MEDIA MODE

Table Notes:

AP - Allow Preemption if arbitration is enabled.

1. 0x20EC bit 0 determines whether the level of preemption is command or object level.

2. 0x20E4 bits 2:1 determine the level of preemption for GPGPU workloads.

3. MI_ATOMIC and MI_SEMAPHORE_SIGNAL commands with Post Sync Op bit set are treated as PIPE_CONTROL command with Post Sync Operation as Atomics or Semaphore Signal.

4. Any Header with the value [31:29] = "011", [28:27] = "00" OR "11" and [26:24] = "001". Refer to **Graphics Command Formats**.

Compute Command Streamer

Enabling Multi context

Multi context mode is enabled by setting the *Compute Engine Enable* bit in the **Render Control Unit Mode** Register.

This is a global control bit and hardware responds to submission of workloads to the Compute CS only when this bit is set (render engine must be idle when programming this bit)

Software Interface

Render CS and the Compute CS have their own independent execution list interfaces that SW can schedule too independently.

As these engines are executing asynchronously:

- Each CS sends its interrupts to both GuC and Host Interrupt block.
- Each CS provides its status independently (using the Context Status structure).

Context Priority

Scheduler SW can set priority of a context by specifying the priority in the context descriptor. Context priority can be changed across submissions - for e.g: submitted as low priority at submission n and then high priority at submission n+1

-- Table of priorities and resulting behavior to be added --

Pre-emption Support

RCS Hardware running in Single-Context Mode

Supported modes of Preemption
• 3D mode: Batch level, Draw (command) level, Object level • GPGPU mode: Thread Group level

RCS and CCS Hardware running in Multi-Context Mode

Supported modes of Preemption
• RCS only supports Thread Group pre-emption. • All compute CS-es only support Thread Group pre-emption.

The following table describes the hardware behavior when pre-emption is invoked in different multi-context scenarios.

Context 0 (R0): Running on RCS	Context1 (C0): Running on COMPUTE CS	Pre-empt Trigger	Behavior
3D	Idle	Preempt R0	Batch/Draw/Object
Compute	Idle	Preempt R0	Thread Group
Idle	Compute	Preempt C0	Thread Group
3D	Compute	Preempt R0	Pre-empt 3D at batch/draw/object
3D	Compute	Preempt C0	Thread group preempt C0
Compute	Compute	Preempt R0	Thread group preempt of R0
Compute	Compute	Preempt C0	Thread group preempt of C0

Resets

When Reset is invoked, the entire render engine is reset.

- This reset impacts all the contexts currently running on the machine.
- Hardware does not support a partial reset - where only one context executing on the render engine is reset.

If the hardware is running multiple contexts and if a context needs to be reset (for e.g because it is hung), then SW has to evict any other normally running context from the machine before it triggers Reset.

Address Space Considerations

Address Space Programming Note	
Context:	Address Space On Skus
Full 48b address space is supported for all contexts (unlike skus where only 47b address space was supported in dual context mode)	

CCS Register State Context

This section summarizes the ComputeCS register state context.

Compute Engine Register State Context

Color Coding

EXECLIST CONTEXT
EXECLIST CONTEXT(PPGTT Base)
ENGINE CONTEXT
EXTENDED ENGINE CONTEXT

Description	MMIO Offset/Command	Unit	# of DW
NOOP		CSEL	1
MI_LOAD_REGISTER_IMM	0x1108_1019	CSEL	1
Context Save/Restore Control Register	0x1A244	CSEL	2
Ring Buffer Head	0x1A034	CSEL	2
Ring Tail Pointer Register	0x1A030	CSEL	2
RING_BUFFER_START	0x1A038	CSEL	2
RING_BUFFER_CONTROL	0x1A03C	CSEL	2
Batch Buffer Current Head Register (UDW)	0x1A168	CSEL	2
Batch Buffer Current Head Register	0x1A140	CSEL	2
Batch Buffer State Register	0x1A110	CSEL	2
BB_PER_CTX_PTR	0x1A1C0	CSEL	2
CS_INDIRECT_CTX	0x1A1C4	CSEL	2
CS_INDIRECT_CTX_OFFSET	0x1A1C8	CSEL	2
CCID	0x1A180	CSEL	2
SEMAPHORE_TOKEN	0x1A2B4	CSEL	2
NOOP		CSEL	4
NOOP		CSEL	1

Description	MMIO Offset/Command	Unit	# of DW
MI_LOAD_REGISTER_IMM	0x1108_1011	CSEL	1
CTX_TIMESTAMP	0x1A3A8	CSEL	2
PDP3_UDW	0x1A28C	CSEL	2
PDP3_LDW	0x1A288	CSEL	2
PDP2_UDW	0x1A284	CSEL	2
PDP2_LDW	0x1A280	CSEL	2
PDP1_UDW	0x1A27C	CSEL	2
PDP1_LDW	0x1A2278	CSEL	2
PDP0_UDW	0x1A274	CSEL	2
PDP0_LDW	0x1A270	CSEL	2
MI_LOAD_REGISTER_IMM	0x1108_0005	CSEL	2
POSH_LRCA (DUMMY)	0x1A1B0	CSEL	2
CONTEXT_SCHEDULING_ATTRIBUTES (RESOURCE_MIN_MAX_PRIROITY)	0x1A5A8	CSEL	2
PREEMPTION_STATUS	0x1A5AC	CSEL	2
NOOP		CSEL	4
NOOP		CSEL	1
MI_LOAD_REGISTER_IMM	0x1108_0001	CSEL	1
R_PWR_CLK_STATE	0x1A0C8	CSEL	2
GPGPU_CSR_BASE_ADDRESS		CSEL	3
NOOP		CSEL	9
MI_LOAD_REGISTER_MEM (INT_MASK_ENABLE from DWORD GFX_ADDR)		CSEL	4
NOOP		CSEL	1
MI_LOAD_REGISTER_IMM	0x1108_1003	CSEL	1
INT_STATUS_REPORT_PTR		CSEL	2
INT_SRC_REPORT_PTR		CSEL	2
NOOP		CSEL	6
NOOP		CSFE	1
MI_LOAD_REGISTER_IMM	0x1108_106D	CSFE	1
BB_STACK_WRITE_PORT	0x1A588	CSFE	12
EXCC	0x1A028	CSFE	2
MI_MODE	0x1A09C	CSFE	2
INSTPM	0x1A0C0	CSFE	2
PR_CTR_CTL	0x1A178	CSFE	2
PR_CTR_THRSH	0x1A17C	CSFE	2

Description	MMIO Offset/Command	Unit	# of DW
TIMESTAMP Register (LSB)	0x1A358	CSFE	2
BB_START_ADDR_UDW	0x1A170	CSFE	2
BB_START_ADDR	0x1A150	CSFE	2
BB_ADD_DIFF	0x1A154	CSFE	2
BB_OFFSET	0x1A158	CSFE	2
MI_PREDICATE_RESULT_1	0x1A41C	CSFE	2
CS_GPR (1-16)	0x1A600	CSFE	64
IPEHR	0x1A068	CSFE	2
CMD_BUF_CCTL	0x1A084	CSFE	2
CS_MI_ADDRESS_OFFSET	0x1A3B4	CSFE	2
MI_SET_PREDICATE_RESULT	0x1A3B8	CSFE	2
WPARID	0x1A21C	CSFE	2
PREDICATION_MASK	0x1A1FC	CSFE	2
NOOP		CSFE	9
MI_LOAD_REGISTER_IMM	0x1102_1008	CSFE	1
TRTTCR	0x4580	CSFE	2
TRVADR	0x4584	CSFE	2
TRTT_L3_BASE_LOW	0x4588	CSFE	2
TRTT_L3_BASE_HIGH	0x458C	CSFE	2
TR_NULL_GFX	0x4590	CSFE	2
TR_INV	0xF594	CSFE	2
NOOP		CSFE	4
NOOP		CSFE	6
NOOP		CSBE	1
MI_LOAD_REGISTER_IMM	0x1108_1023	CSBE	1
CS_CONTEXT_STATUS1	0x1A184	CSBE	2
GPUGPU_DISPATCHDIMX	0x1A500	CSBE	2
GPUGPU_DISPATCHDIMY	0x1A504	CSBE	2
GPUGPU_DISPATCHDIMZ	0x1A508	CSBE	2
MI_PREDICATE_SRC0	0x1A400	CSBE	2
MI_PREDICATE_SRC0	0x1A404	CSBE	2
MI_PREDICATE_SRC1	0x1A408	CSBE	2
MI_PREDICATE_SRC1	0x1A40C	CSBE	2
MI_PREDICATE_DATA	0x1A410	CSBE	2
MI_PREDICATE_DATA	0x1A414	CSBE	2
MI_PRED_RESULT	0x1A418	CSBE	2

Description	MMIO Offset/Command	Unit	# of DW
GPGPU_THREADS_DISPATCHED	0x1A290	CSBE	4
NOOP		CSBE	4
NOOP		CSBE	2
PIPELINE_SELECT		CSBE	1
STATE_BASE_ADDRESS		CSBE	22
STATE_SIP		CSBE	3
STATE_COMPUTE_MODE		CSBE	2
3DSTATE_BINDING_TABLE_POOL_ALLOC		CSBE	4
3DSTATE_BTD		CSBE	6
NOOP		CSBE	8
NOOP		CSBE	6
MI_BATCH_BUFFER_END		CSEND	1
NOOP		CSEND	127

CCS Power Context Image

This section summarizes the ComputeCS power context.

Compute Engine Power Context

Compute Engine Power Context Image

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
CSFE Power context without Display		CSFE	195	**0**	**CSFE**
NOOP		**CS**	**1**	**00C4**	**CSBE**
Load_Register_Immediate header	**0x1100_100F**	**CS**	**1**		**CSBE**
NOOP		CS	4		CSBE
NOOP		**CS**	**1**	**00DA**	**CSBE**
MI_BATCH_BUFFER_END		**CS**	**1**	**00DB**	**CSBE**

MI Commands Supported by ComputeCS

ComputeCS supports all the MI commands supported by RCS except for the below exceptions.

ComputeCS doesn't support below commands and SW must not program them as part of the Compute engine's command sequence.

Commands not supported in ComputeCS executed command buffers:

Commands
MI_DISPLAY_FLIP
MI_LOAD_SCANLINES_INCL/EXCL
MI_WAIT_FOR_EVENT
MI_FORCE_WAKEUP

GPGPU-Media State Commands Supported by ComputeCS

The table below lists the GPGPU-MEDIA STATE Commands Supported by Compute Engine (ComputeCS). State commands programmed for ComputeEngine which are not listed in the table below will be gracefully discarded (NOOP'd) by ComputeCS.

GPGPU-MEDIA State Commands
• CFE_STATE • COMPUTE_WALKER
Common Non-Pipeline State Commands
• STATE_BASE_ADDRESS • GPGPU_CSR_BASE_ADDRESS • PIPELINE_SELECT • STATE_SIP
• STATE_COMPUTE_MODE
• 3DSTATE_BTD
Misc Commands
• PIPE_CONTROL • 3DSTATE_BINDING_TABLE_POOL_ALLOC
• L3_CONTROL

Graphics Pipeline Software Interface

This section covers the Graphics pipeline software interface.

Synchronization of the Graphics Pipeline

Two types of synchronizations are supported for the 3D pipe: top of the pipe and end of the pipe. Top of the pipe synchronization really enforces the read-only cache invalidation. This synchronization

guarantees that primitives rendered after such synchronization event fetches the latest read-only data from memory. End of the pipe synchronization enforces that the read and/or read-write buffers do not have outstanding hardware accesses. These are used to implement read and write fences as well as to write out certain statistics deterministically with respect to progress of primitives through the pipeline (and without requiring the pipeline to be flushed.) The PIPE_CONTROL command (see details below) is used to perform all above synchronizations.

Top-of-Pipe Synchronization

Top-of-pipe synchronization refers to SW actions to prepare HW for new state-binding at the beginning of the rendering sequence in a given context. HW may have residual states cached in the state-caches and read-only surfaces in various caches. With new rendering sequence, read-only surfaces may go through change in the binding. Hence read-only invalidation is required before such new rendering sequence. Read-only cache invalidation is top-of-pipe synchronization. Upon parsing this specific pipe-control command, HW invalidates all caches in GT domain that have read-only surfaces but does not guarantee invalidation beyond GT caches

Upon parsing this specific pipe-control command, HW invalidates all caches in GT domain that have read-only surfaces but does not guarantee invalidation beyond GT caches (i.e. LLC).

Further, HW does not guarantee that all prior accesses to those read-only surfaces have completed. Therefore, SW must guarantee that there are no pending accesses to those read-only surfaces before initializing the top-of-pipe synchronization. PIPE-CONTROL command described below allows for invalidating individual read-only stream type. It is recommended that driver invalidates only the required caches on the need basis so that cache warm-up overhead can be reduced.

End-of-Pipe Synchronization

The driver can use end-of-pipe synchronization to know that rendering is complete (although not necessarily in memory) so that it can deallocate in-memory rendering state, read-only surfaces, instructions, and constant buffers. An end-of-pipe synchronization point is also sufficient to guarantee that all pending depth tests have completed so that the visible pixel count is complete prior to storing it to memory. End-of-pipe completion is sufficient (although not necessary) to guarantee that read events are complete (a "read fence" completion). Read events are still pending if work in the pipeline requires any type of read except a render target read (blend) to complete.

Write synchronization is a special case of end-of-pipe synchronization that requires that the render cache and/or depth related caches are flushed to memory, where the data will become globally visible. This type of synchronization is required prior to SW (CPU) actually reading the result data from memory or initiating an operation that will use as a read surface (such as a texture surface) a previous render target and/or depth/stencil buffer. Exercising the write cache flush bits (Render Target Cache Flush Enable, Depth Cache Flush Enable, DC Flush) in PIPE_CONTROL only ensures the write caches are flushed and doesn't guarantee the data is globally visible.

SW can track the completion of the end-of-pipe-synchronization by using "Notify Enable" and "Post-Sync Operation - Write Immediate Data" in the PIPE_CONTROL command. "Notify Enable" and "Post-Sync Operation - Write Immediate Data" generate a fence cycle on achieving end-of-pipe-

synchronization for the corresponding PIPE_CONTROL command. Fence cycle ensures all the write cycles in front of it are to global visible point before they themselves get processed. It is guaranteed the data flushed out by the PIPE_CONTROL is updated in memory by the time SW receives the corresponding Pipe Control Notify interrupt.

In case the data flushed out by the render engine is to be read back into the render engine in coherent manner, then the render engine has to wait for the fence completion before accessing the flushed data. This can be achieved by following means on various products.

PIPE_CONTROL command with CS Stall and the required write caches flushed with Post-Sync-Operation as Write Immediate Data.

Example:

- WorkLoad-1 (3D/GPGPU/MEDIA)
- PIPE_CONTROL (CS Stall, Post-Sync-Operation Write Immediate Data, Required Write Cache Flush bits set)
- WorkLoad-2 (Can use the data produced or output by Workload-1)

Synchronization Actions

In order for the driver to act based on a synchronization point (usually the whole point), the reaching of the synchronization point must be communicated to the driver. This section describes the actions that may be taken upon completion of a synchronization point which can achieve this communication.

Writing a Value to Memory

The most common action to perform upon reaching a synchronization point is to write a value out to memory. An immediate value (included with the synchronization command) may be written. In lieu of an immediate value, the 64-bit value of the PS_DEPTH_COUNT (visible pixel count) or TIMESTAMP register may be written out to memory. The captured value will be the value at the moment all primitives parsed prior to the synchronization commands have been completely rendered, and optionally after all said primitives have been pushed to memory. It is not required that a value be written to memory by the synchronization command.

Visible pixel or TIMESTAMP information is only useful as a delta between 2 values, because these counters are free-running and are not to be reset except at initialization. To obtain the delta, two PIPE_CONTROL commands should be initiated with the command sequence to be measured between them. The resulting pair of values in memory can then be subtracted to obtain a meaningful statistic about the command sequence.

PS_DEPTH_COUNT

If the selected operation is to write the visible pixel count (PS_DEPTH_COUNT register), the synchronization command should include the **Depth Stall Enable** parameter. There is more than one point at which the global visible pixel count can be affected by the pipeline; once the synchronization command reaches the first point at which the count can be affected, any primitives following it are stalled at that point in the pipeline. This prevents the subsequent primitives from affecting the visible

pixel count until all primitives preceding the synchronization point reach the end of the pipeline, the visible pixel count is accurate, and the synchronization is completed. This stall has a minor effect on performance and should only be used in order to obtain accurate "visible pixel" counts for a sequence of primitives.

The PS_DEPTH_COUNT count can be used to implement an (API/DDI) "Occlusion Query" function.

Generating an Interrupt

The synchronization command may indicate that a "Sync Completion" interrupt is to be generated (if enabled by the MI Interrupt Control Registers - see *Memory Interface Registers*) once the rendering of all prior primitives is complete. Again, the completion of rendering can be considered to be when the internal render cache has been updated, or when the cache contents are visible in memory, as selected by the command options.

Invalidating of Caches

If software wishes to use the notification that a synchronization point has been reached in order to reuse referenced structures (surfaces, state, or instructions), it is not sufficient just to make sure rendering is complete. If additional primitives are initiated after new data is laid over the top of old in memory following a synchronization point, it is possible that stale cached data will be referenced for the subsequent rendering operation. In order to avoid this, the PIPE_CONTROL command must be used. (See PIPE_CONTROL Command description).

PIPE_CONTROL Command

The PIPE_CONTROL command provides mechanism to achieve the synchronization of the 3D pipeline and to execute post-synchronization operations as described in the section "Synchronization of the 3D pipeline". Parsing a PIPE_CONTROL command stalls the 3D pipe only if the stall enable bit is set. Commands after PIPE_CONTROL will continue to be parsed and processed in the 3D pipeline. This may include additional PIPE_CONTROL commands. The implementation does enforce a practical upper limit (8) on the number of PIPE_CONTROL commands that may be outstanding at once. Parsing a PIPE_CONTROL command that causes this limit to be reached will stall the parsing of new commands until the first of the outstanding PIPE_CONTROL commands reaches the end of the pipe and retires.

Although PIPE_CONTROL is intended for use with the 3D pipe, it is legal to issue PIPE_CONTROL when the Media pipe is selected. In this case PIPE_CONTROL will stall at the top of the pipe until the Media FFs finish processing commands parsed before PIPE_CONTROL. Post-synchronization operations, flushing of caches and interrupts will then occur if enabled via PIPE_CONTROL parameters. Due to this stalling behavior, only one PIPE_CONTROL command can be outstanding at a time on the Media pipe.

For the invalidate operation of the pipe control, the following pointers are affected. The invalidate operation affects the context restore of these packets. If the pipe control invalidate operation is completed before the context save, the indirect pointers will not be context restored from memory on a context switch.

- Pipeline State Pointer

- Media State Pointer
- Constant Buffer Packet
- SW must ensure to invalidate the Media State and Constant Buffers using "Generic Media State Clear" prior to the releasing the associated resources (memory).
- SW must ensure to invalidate the Push Constants using "Indirect State Pointers Disable" prior to the releasing the associated resources (memory).

It is up to software to program the appropriate read-only cache invalidation such as the sampler and constant read caches or the instruction and state caches. Once notification is observed, new data may then be loaded (potentially "on top of" the old data) without fear of stale cache data being referenced for subsequent rendering.

If software wishes to access the rendered data in memory (for analysis by the application or to copy it to a new location to use as a texture, for example), it must also ensure that the write cache (render cache) is flushed after the synchronization point is reached so that memory will be updated. This can be done by setting the **Write Cache Flush Enable** bit. Note that the **Depth Stall Enable** bit must be clear in order for the flush of the render cache to occur. **Depth Stall Enable** is intended only for accurate reporting of the PS_DEPTH counter; the render cache cannot be flushed nor can the read caches be invalidated (except for the instruction/state cache) in conjunction with this operation.

Vertex caches are only invalidated when the VF invalidate bit is set in PIPE_CONTROL (i.e. decision is done in software, not hardware) Note that the index-based vertex cache is always flushed between primitive topologies and of course PIPE_CONTROL can only be issued between primitive topologies. Therefore only the VF ("address-based") cache is uniquely affected by PIPE_CONTROL.

PIPE_CONTROL

PIPE_CONTROL

Hardware supports up to 32 pending PIPE_CONTROL flushes.

The table below explains all the different flush/invalidation scenarios.

Caches Invalidated/Flushed by PIPE_CONTROL Bit Settings

Write Cache Flush	Notification Enabled	Non-VF RO Cache Invalidate	VF RO Cache Invalidate	Marker Sent	Pipeline Marker Enable	Completion Requested	Top of Pipe Invalidate Pulse from CS
0	0	0	0	N/A	N/A	N/A	N/A
0	0	0	1	Yes	No	N/A	No
0	0	1	0	No	N/A	N/A	Yes
0	0	1	1	Yes	No	No	Yes
X	1	0	X	Yes	Yes	Yes	No
X	1	1	X	Yes	Yes	Yes	Yes
1	X	0	X	Yes	Yes	Yes	No
1	X	1	X	Yes	Yes	Yes	Yes

Programming Restrictions for PIPE_CONTROL

PIPE_CONTROL arguments can be split up into three categories:

- Post-sync operations
- Flush Types
- Stall

Post-sync operation is only indirectly affected by the flush type category via the stall bit. The stall category depends on both flush type and post-sync operation arguments. A PIPE_CONTROL with no arguments set is **Invalid**.

Post-Sync Operation

These arguments relate to events that occur after the marker initiated by the PIPE_CONTROL command is completed. The table below shows the restrictions:

Argument	Bits	Restriction
LRI Post Sync Operation	23	Post Sync Operation ([15:14] of DW1) must be set to 0x0.
Global Snapshot Count Reset	19	This bit must not be exercised on any product. Requires stall bit ([20] of DW1) set.
Generic Media State Clear	16	Requires stall bit ([20] of DW1) set.
Generic Media State Clear	16	Must not be set in PIPECONTROL command programmed for POCS.
Indirect State Pointers Disable	9	Requires stall bit ([20] of DW1) set.
Store Data Index	21	Post-Sync Operation ([15:14] of DW1) must be set to something other than '0'.
Sync GFDT	17	Post-Sync Operation ([15:14] of DW1) must be set to something other than '0' or 0x2520[13] must be set.
TLB inv	18	Requires stall bit ([20] of DW1) set.
Post Sync Op	15:14	LRI Post Sync Operation ([23] of DW1) must be set to '0'.
Post Sync Op	15:14	Post Sync Operations must not be set to "Write PS Depth Count" in PIPECONTROL command programmed for POCS.
Notify En	8	Must not be set in PIPECONTROL command programmed for POCS.

Flush Types

These are arguments related to the type of read only invalidation or write cache flushing is being requested. Note that there is only intra-dependency. That is, it is not affected by the post-sync operation or the stall bit. The table below shows the restrictions:

Arguments	Bit	Restrictions

Tile Cache Flush	28	**Tile Cache Enabled Mode:**
		• SW must always set CS Stall bit when Tile Cache Flush Enable bit is set in the PIPECONTROL command.
		• SW must ensure level1 depth and color caches are flushed prior to flushing the tile cache. This can be achieved by following means:
		  ○ Single PIPECONTROL command to flush level1 caches and the tile cache. Hardware will sequence the flushing of L1 caches followed by the Tile cache. Attributes listed below must be set. OR
		  ▪ Tile Cache Flush Enable
		  ▪ Render Target Cache Flush Enable
		  ▪ Depth Cache Flush Enable
		  ○ Flushing of L1 caches followed by flushing of tile cache through two different PIEPCONTROL commands. SW must ensure not to issue any rendering commands between the two PIPECONTROL commands.
		Unified Cache (Tile Cache Disabled):
		In unified cache mode of operation Color and Depth (Z) streams are cached in DC space of L2 along with Data Port stream.
		On a "Tile Cache Flush" only Color and Depth (Z) streams from DC space of L2 are flushed to globally observable and where as "DC Flush Enable" will only flush Data Port stream from the DC space of L2 to globally observable. Refer L3 configuration section for Unified cache usage model. In this mode of operation there is no dedicated memory allocated for Tile Cache in L2.
		When the Color and Depth (Z) streams are enabled to be cached in the DC space of L2, Software must use "Render Target Cache Flush Enable" and "Depth Cache Flush Enable" along with "Tile Cache Flush" for getting the color and depth (Z) write data to be globally observable. In this mode of operation it is not required to set "CS Stall" upon setting "Tile Cache Flush" bit.

Depth Stall		Must not set in PIPECONTROL command programmed for POCS.
Render Target Cache Flush		Must not be set in PIPECONTROL command programmed for POCS.
Depth Cache Flush		Must not be set in PIPECONTROL command programmed for POCS.
Stall Pixel Scoreboard	1	No Restriction.

Arguments	Bit	Restrictions
Stall Pixel Scoreboard		Must not be set in PIPECONTROL command programmed for POCS.
DC Flush Enable		Must not be set in PIPECONTROL command programmed for POCS.
Inst invalidate	11	No Restriction.
Tex invalidate	10	Requires stall bit ([20] of DW) set for all GPGPU Workloads.
Constant invalidate	3	No Restriction. This bit also invalidates the Ray Tracing (BVH Data) Cache.
State Invalidate	2	No Restriction.

Stall

If the stall bit is set, the command streamer waits until the pipe is completely flushed.

Arguments	Bit	Restrictions
Stall Bit	20	No Restrictions.

Engine State

This section describes the state specific to the Graphics Engine

Memory Access Indirection

The GPE supports the indirection of certain graphics (GTT-mapped) memory accesses. This support comes in the form of two *base address* state variables used in certain memory address computations with the GPE.

The intent of this functionality is to support the dynamic relocation of certain driver-generated memory structures after command buffers have been generated but prior to their submittal for execution. For example, as the driver builds the command stream it could append pipeline state descriptors, kernel binaries, etc. to a general state buffer. References to the individual items would be inserted in the command buffers as offsets from the base address of the state buffer. The state buffer could then be freely relocated prior to command buffer execution, with the driver only needing to specify the final base address of the state buffer. Two base addresses are provided to permit surface-related state (binding tables, surface state tables) to be maintained in a state buffer separate from the general state buffer.

While the use of these base addresses is unconditional, the indirection can be effectively disabled by setting the base addresses to zero. The following table lists the various GPE memory access paths and which base address (if any) is relevant.

Base Address Utilization

Base Address Used	Memory Accesses
General State Base Address	**DataPort Read/Write DataPort memory accesses resulting from 'stateless' DataPort Read/Write requests. See** *DataPort* **for a definition of the 'stateless' form of requests.**
Dynamic State Base Address	Sampler reads of SAMPLER_STATE data and associated SAMPLER_BORDER_COLOR_STATE
Dynamic State Base Address	**Viewport states** used by CLIP, SF, and WM/CC
Dynamic State Base Address	COLOR_CALC_STATE, DEPTH_STENCIL_STATE, and BLEND_STATE
Dynamic State Base Address	Push Constants (depending on state of **INSTPM<CONSTANT_BUFFER Address Offset Disable>**)
Instruction Base Address	**Normal EU instruction stream** (non-system routine)
Instruction Base Address	**System routine** EU instruction stream (starting address = SIP)
Surface State Base Address	Sampler and DataPort reads of BINDING_TABLE_STATE, as referenced by BT pointers passed via 3DSTATE_BINDING_TABLE_POINTERS
Surface State Base Address	Sampler and DataPort reads of SURFACE_STATE data
None	CS unit reads from **Ring Buffers**, **Batch Buffers**
None	CS writes resulting from PIPE_CONTROL command
None	All VF unit memory accesses (**Index Buffers**, **Vertex Buffers**)
None	All Sampler **Surface Memory Data** accesses (texture fetch, etc.)
None	All **DataPort memory accesses** except 'stateless' DataPort Read/Write requests (e.g., RT accesses). See *DataPort* for a definition of the 'stateless' form of requests.
None	Memory reads resulting from **STATE_PREFETCH** commands
None	Any **physical memory access** by the device

Base Address Used	Memory Accesses
None	GTT-mapped accesses not included above (i.e., default)
None	Push Constants (depending on state of **INSTPM<CONSTANT_BUFFER Address Offset Disable>**)

The following notation is used to distinguish between addresses and offsets:

Notation	Definition
PhysicalAddress[n:m]	Corresponding bits of a physical graphics memory byte address (not mapped by a GTT)
GraphicsAddress[n:m]	Corresponding bits of an absolute, virtual graphics memory byte address (mapped by a GTT)
GeneralStateOffset[n:m]	Corresponding bits of a relative byte offset added to the General State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT)
DynamicStateOffset[n:m]	Corresponding bits of a relative byte offset added to the Dynamic State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT)
InstructionBaseOffset[n:m]	Corresponding bits of a relative byte offset added to the Instruction Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT)
SurfaceStateOffset[n:m]	Corresponding bits of a relative byte offset added to the Surface State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT)

Context Image

Logical Contexts are memory images used to store copies of the device's rendering and ring context.

Logical Contexts are aligned to 256-byte boundaries.

Logical contexts are referenced by their memory address. The format and contents of rendering contexts are considered ***device-dependent*** and software must not access the memory contents directly. The definition of the logical rendering and power context memory formats is included here primarily for internal documentation purposes.

Power Context Image

Render Engine Power Context

RCS Power Context Image

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
CSFE Power Context with Display			208	0	CSFE
NOOP		CS	1	00D0	CSBE
Load_Register_Immediate header	0x1100_0019	CS	1	00D1	CSBE
GAFS_Mode	0x212C	CS	2	00DC	CSBE
Dummy(Defeatured)	0x24C4	CS	2	00E2	CSBE
Dummy(Defeatured)	0x24C0	CS	2	00E4	CSBE
FF_STALL_CHK_VFUNIT	0x83B0	VF	2	00E8	CSBE
VFGPCB_VFUNIT	0x83B4	VF	2	00EA	CSBE
NOOP		CS	3	00EC	CSBE
MI_BATCH_BUFFER_END		CS	1	00EF	CSBE

Compute Engine Power Context

Compute Engine Power Context Image

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
CSFE Power context without Display		CSFE	195	0	CSFE
NOOP		CS	1	00C4	CSBE
Load_Register_Immediate header	0x1100_0009	CS	1	00C5	CSBE
NOOP		CS	10	00D0	CSBE
NOOP		CS	1	00DA	CSBE
MI_BATCH_BUFFER_END		CS	1	00DB	CSBE

The table below captures the data from TDL power context save/restored by PM.

TDL Power Context Image

Description	Offset	Unit	# of DW
NOOP		TDL	1
MI_LOAD_REGISTER_IMM	0x1100_0057	TDL	1
TD_CTL	E400	TDL	2
TD_CTL2	E404	TDL	2
TD_VF_VS_EMSK	E408	TDL	2
TD_GS_EMSK	E40C	TDL	2
TD_WIZ_EMSK	E410	TDL	2
TD_TS_EMSK	E428	TDL	2
TD_HS_EMSK	E4B0	TDL	2
TD_DS_EMSK	E4B4	TDL	2
EU_PERF_CNT_CTL0	E458	TDL	2
EU_PERF_CNT_CTL1	E558	TDL	2
EU_PERF_CNT_CTL2	E658	TDL	2
EU_PERF_CNT_CTL3	E758	TDL	2
EU_PERF_CNT_CTL4	E45C	TDL	2
EU_PERF_CNT_CTL5	E55C	TDL	2
EU_PERF_CNT_CTL6	E65C	TDL	2
CULLBIT3	E488	TDL	2
CACHE_MODE_SS	E420	TDL	2
RAY TRACING CONTROL	E530	TDL	2
VSR_EMASK	E51C	TDL	2
SLM_BANK_HASH	E660	TDL	2
SLM_ECC_ERROR_CNT	E7F4	TDL	2
SLM_UNCORR_ECC_ERROR_CNT	E7C0	TDL	2
RT_CTRL	E530	TDL	2
NOOP		TDL	1
MI_BATCH_BUFFER_END		TDL	1

Sampler Power Context

The table below captures the data from TDL power context save/restored by PM.

Sampler Power Context Image

Description	Offset	Unit	# of DW
NOOP		SC	1
MI_LOAD_REGISTER_IMM	0x1100_000D	SC	1
SAMPLER_MODE	0xE18C	SC	2

Engine Register and State Context

This section describes programming requirements for the Register State Context.

EXECLIST CONTEXT(Ring)
EXECLIST CONTEXT (PPGTT Base)
ENGINE CONTEXT(CSFE)
ENGINE CONTEXT(CSBE)
ENGINE CONTEXT(SOL)
ENGINE CONTEXT(VF)
ENGINE CONTEXT(GAMWC)
ENGINE CONTEXT(GAMT)
ENGINE CONTEXT(LNCF)
ENGINE CONTEXT(SVG)
ENGINE CONTEXT(SVL)
ENGINE CONTEXT(TDL)
ENGINE CONTEXT(WM)
ENGINE CONTEXT(SC)
ENGINE CONTEXT(DM)
ENGINE CONTEXT(VFE)
ENGINE CONTEXT(CS - Footer)

EXECLIST CONTEXT(Ring)
EXECLIST CONTEXT (PPGTT Base)
ENGINE CONTEXT(CSFE)
ENGINE CONTEXT(CSBE)
ENGINE CONTEXT(SOL)
ENGINE CONTEXT(VF)
ENGINE CONTEXT(OAR)

ENGINE CONTEXT(LBCF)
ENGINE CONTEXT(SVG)
ENGINE CONTEXT(SVL)
ENGINE CONTEXT(WM)
ENGINE CONTEXT(SC)
ENGINE CONTEXT(VFE)
ENGINE CONTEXT(CS - Footer)

ComputeCS Context Image

EXECLIST CONTEXT(Ring)
EXECLIST CONTEXT (PPGTT Base)
ENGINE CONTEXT(CSFE)
ENGINE CONTEXT(CSBE)
ENGINE CONTEXT(VFE)
ENGINE CONTEXT(CS - Footer)

POSH Context Image

EXECLIST CONTEXT(Ring)
EXECLIST CONTEXT (PPGTT Base)
ENGINE CONTEXT(CSFE)
ENGINE CONTEXT(CSBE)
ENGINE CONTEXT(VFR)
ENGINE CONTEXT(OVR)
ENGINE CONTEXT(SVGR)
ENGINE CONTEXT(CS - Footer)

Register State Context

Color Coding

EXECLIST CONTEXT
EXECLIST CONTEXT(PPGTT Base)
ENGINE CONTEXT
EXTENDED ENGINE CONTEXT

Description	MMIO Offset/Command	Unit	# of DW
NOOP		CSEL	1
MI_LOAD_REGISTER_IMM	0x1108_101D	CSEL	1
Ring Buffer Head	0x2034	CSEL	2
Ring Tail Pointer Register	0x2030	CSEL	2
RING_BUFFER_START	0x2038	CSEL	2
RING_BUFFER_CONTROL	0x203C	CSEL	2
Batch Buffer Current Head Register (UDW)	0x2168	CSEL	2
Batch Buffer Current Head Register	0x2140	CSEL	2
Batch Buffer State Register	0x2110	CSEL	2
BB_PER_CTX_PTR	0x21C0	CSEL	2
RCS_INDIRECT_CTX	0x21C4	CSEL	2
RCS_INDIRECT_CTX_OFFSET	0x21C8	CSEL	2
CCID	0x2180	CSEL	2
SEMAPHORE_TOKEN	0x22B4	CSEL	2
PRT_BB_STATE	0x2120	CSEL	4
NOOP		CSEL	1
MI_LOAD_REGISTER_IMM	0x1100_1011	CSEL	1
CTX_TIMESTAMP	0x23A8	CSEL	2
PDP3_UDW	0x228C	CSEL	2
PDP3_LDW	0x2288	CSEL	2
PDP2_UDW	0x2284	CSEL	2
PDP2_LDW	0x2280	CSEL	2
PDP1_UDW	0x227C	CSEL	2
PDP1_LDW	0x2278	CSEL	2
PDP0_UDW	0x2274	CSEL	2
PDP0_LDW	0x2270	CSEL	2
MI_LOAD_REGISTER_IMM	0x1108_1005	CSEL	1

Description	MMIO Offset/Command	Unit	# of DW
POSH_LRCA	0x21B0	CSEL	2
CONTEXT_SCHEDULING_ATTRIBUTES (RESOURCE_MIN_MAX_PRIORITY)	0x25A8	CSEL	2
PREEMPTION_STATUS (RO)	0x25AC	CSEL	2
NOOP		CSEL	5
NOOP		CSEL_BE	1
MI_LOAD_REGISTER_IMM	0x1108_0001	CSEL_BE	1
R_PWR_CLK_STATE	0x20C8	CSEL_BE	2
GPGPU_CSR_BASE_ADDRESS		CSEL_BE	3
NOOP		CSEL_BE	9
MI_LOAD_REGISTER_MEM (INT_MASK_ENABLE from DWORD GFX_ADDR)		CSEL	4
NOOP		CSEL	1
MI_LOAD_REGISTER_IMM	0x1108_1003	CSEL	1
INT_STATUS_REPORT_PTR		CSEL	2
INT_SRC_REPORT_PTR		CSEL	2
NOOP		CSEL	6
NOOP		CSFE	1
MI_LOAD_REGISTER_IMM	0x1108_106D	CSFE	1
BB_STACK_WRITE_PORT	0x2588	CSFE	12
EXCC	0x2028	CSFE	2
MI_MODE	0x209C	CSFE	2
INSTPM	0x20C0	CSFE	2
PR_CTR_CTL	0x2178	CSFE	2
PR_CTR_THRSH	0x217C	CSFE	2
TIMESTAMP Register (LSB)	0x2358	CSFE	2
BB_START_ADDR_UDW	0x2170	CSFE	2
BB_START_ADDR	0x2150	CSFE	2
BB_ADD_DIFF	0x2154	CSFE	2
BB_OFFSET	0x2158	CSFE	2
MI_PREDICATE_RESULT_1	0x241C	CSFE	2
CS_GPR (1-16)	0x2600	CSFE	64
IPEHR	0x2068	CSFE	2
CMD_BUF_CCTL	0x2084	CSFE	2
CS_MI_ADDRESS_OFFSET	0x23B4	CSFE	2
MI_SET_PREDICATE_RESULT	0x23B8	CSFE	2

Description	MMIO Offset/Command	Unit	# of DW
WPARID	0x221C	CSFE	2
PREDICATION_MASK	0x21FC	CSFE	2
NOOP		CSFE	9
MI_LOAD_REGISTER_IMM	0x1102_100B	CSFE	1
TRTT_CR	0x4400	CSFE	2
TRTT_VA_RANGE	0x4404	CSFE	2
TRTT_L3_BASE_LOW	0x4408	CSFE	2
TRTT_L3_BASE_HIGH	0x440C	CSFE	2
TR_NULL_GFX	0x4410	CSFE	2
TRTT_INVAL	0x4414	CSFE	2
NOOP		CSFE	11
MI_LOAD_REGISTER_IMM	0x1100_10BD	CSBE	1
CS_CONTEXT_STATUS1	0x2184	CSBE	2
IA_VERTICES_COUNT	0x2310	CSBE	4
IA_PRIMITIVES_COUNT	0x2318	CSBE	4
VS_INVOCATION_COUNT	0x2320	CSBE	4
HS_INVOCATION_COUNT	0x2300	CSBE	4
DS_INVOCATION_COUNT	0x2308	CSBE	4
GS_INVOCATION_COUNT	0x2328	CSBE	4
GS_PRIMITIVES_COUNT	0x2330	CSBE	4
CL_INVOCATION_COUNT	0x2338	CSBE	4
CL_PRIMITIVES_COUNT	0x2340	CSBE	4
MESH_INVOCATION_COUNT	0x26E0	CSBE	4
TASK_INVOCATION_COUNT	0x26E8	CSBE	4
MESH_PRIMITIVE_COUNT	0x26D8	CSBE	4
PS_INVOCATION_COUNT_0	0x22C8	CSBE	4
PS_DEPTH_COUNT _0	0x22D8	CSBE	4
GPUGPU_DISPATCHDIMX	0x2500	CSBE	2
GPUGPU_DISPATCHDIMY	0x2504	CSBE	2
GPUGPU_DISPATCHDIMZ	0x2508	CSBE	2
MI_PREDICATE_SRC0	0x2400	CSBE	2
MI_PREDICATE_SRC0	0x2404	CSBE	2
MI_PREDICATE_SRC1	0x2408	CSBE	2
MI_PREDICATE_SRC1	0x240C	CSBE	2
MI_PREDICATE_DATA	0x2410	CSBE	2
MI_PREDICATE_DATA	0x2414	CSBE	2

Description	MMIO Offset/Command	Unit	# of DW
MI_PRED_RESULT	0x2418	CSBE	2
3DPRIM_END_OFFSET	0x2420	CSBE	2
3DPRIM_START_VERTEX	0x2430	CSBE	2
3DPRIM_VERTEX_COUNT	0x2434	CSBE	2
3DPRIM_INSTANCE_COUNT	0x2438	CSBE	2
3DPRIM_START_INSTANCE	0x243C	CSBE	2
3DPRIM_BASE_VERTEX	0x2440	CSBE	2
Load Indirect Extended Parameter 0	0x2690	CSBE	2
Load Indirect Extended Parameter 1	0x2694	CSBE	2
Load Indirect Extended Parameter 2	0x2698	CSBE	2
3DMESH_TG_COUNT	0x26F0	CSBE	2
3DMESH_STARTING_TGID	0x26F4	CSBE	2
GPGPU_THREADS_DISPATCHED	0x2290	CSBE	4
PS_INVOCATION_COUNT_1	0x22F0	CSBE	4
PS_DEPTH_COUNT _1	0x22F8	CSBE	4
RS_PREEMPT_STATUS	0x215C	CSBE	2
PRODUCE_COUNT_BTP	0x2480	CSBE	2
PRODUCE_COUNT_DX9_CONSTANTS	0x2484	CSBE	2
PARSED_COUNT_BTP	0x2490	CSBE	2
PARSED_COUNT_DX9_CONSTANTS	0x2494	CSBE	2
OA_CTX_CONTROL	0x2360	CSBE	2
OACTXID	0x2364	CSBE	2
PS_INVOCATION_COUNT_2	0x2448	CSBE	4
PS_DEPTH_COUNT_2	0x2450	CSBE	4
RS_PREEMPT_STATUS_UDW	0x2174	CSBE	2
CPS_INVOCATION_COUNT	0x2478	CSBE	4
PS_INVOCATION_COUNT_3	0x2458	CSBE	4
PS_DEPTH_COUNT_3	0x2460	CSBE	4
PS_INVOCATION_COUNT_4	0x2468	CSBE	4
PS_DEPTH_COUNT_4	0x2470	CSBE	4
PS_INVOCATION_COUNT_5	0x24A0	CSBE	4
PS_DEPTH_COUNT_5	0x24A8	CSBE	4
PS_INVOCATION_COUNT_6	0x25D0	CSBE	4
PS_DEPTH_COUNT_6	0x25B0	CSBE	4
PS_INVOCATION_COUNT_7	0x25D8	CSBE	4
PS_DEPTH_COUNT_7	0x25B8	CSBE	4

Description	MMIO Offset/Command	Unit	# of DW
NOOP		CSBE	1
MI_LOAD_REGISTER_IMM	0x1100_0001	CSBE	
VSR_PUSHCONSTANT_BASE	0xE518	CSBE	2
NOOP		CSBE	2
MI_TOPOLOGY_FILTER		CSBE	1
NOOP		CSBE	2
PIPELINE_SELECT		CSBE	1
STATE_BASE_ADDRESS		CSBE	22
STATE_SIP		CSBE	3
3DSTATE_PUSH_CONSTANT_ALLOC_VS		CSBE	2
3DSTATE_PUSH_CONSTANT_ALLOC_HS		CSBE	2
3DSTATE_PUSH_CONSTANT_ALLOC_DS		CSBE	2
3DSTATE_PUSH_CONSTANT_ALLOC_GS		CSBE	2
3DSTATE_PUSH_CONSTANT_ALLOC_PS		CSBE	2
3DSTATE_BINDING_TABLE_POOL_ALLOC		CSBE	4
3DSTATE_PTBR_TILE_PASS_INFO		CSBE	4
STATE_COMPUTE_MODE		CSBE	2
3DSTATE_3D_MODE		CSBE	5
3DSTATE_BTD		CSBE	6
NOOP		CSBE	2
NOOP		CSBE	12
NOOP		SOL	1
MI_LOAD_REGISTER_IMM	0x1100_1027	SOL	1
SO_NUM_PRIMS_WRITTEN0	0x5200	SOL	4
SO_NUM_PRIMS_WRITTEN1	0x5208	SOL	4
SO_NUM_PRIMS_WRITTEN2	0x5210	SOL	4
SO_NUM_PRIMS_WRITTEN3	0x5218	SOL	4
SO_PRIM_STORAGE_NEEDED0	0x5240	SOL	4
SO_PRIM_STORAGE_NEEDED1	0x5248	SOL	4
SO_PRIM_STORAGE_NEEDED2	0x5250	SOL	4
SO_PRIM_STORAGE_NEEDED3	0x5258	SOL	4
SO_WRITE_OFFSET0	0x5280	SOL	2
SO_WRITE_OFFSET1	0x5284	SOL	2
SO_WRITE_OFFSET2	0x5288	SOL	2
SO_WRITE_OFFSET3	0x528C	SOL	2
3DSTATE_SO_BUFFER		SOL	32

Description	MMIO Offset/Command	Unit	# of DW
NOOP		SOL	3
3DSTATE_SO_DECL_LIST		SOL	259
NOOP		SOL	0
3DSTATE_INDEX_BUFFER		VF	5
3DSTATE_VERTEX_BUFFERS		VF	133
3DSTATE_VF_STATISTICS		VF	1
3DSTATE_VF		VF	2
3DSTATE_VFG		VF	4
3DSTATE_VF_INSTANCING		VF	69
3DSTATE_VF_TOPOLOGY		VF	2
3DSTATE_MESH_DISTRIB		VF	2
NOOP		VF	6
NOOP		VF	8
MI_LOAD_REGISTER_IMM	0x1100_102B	VF	1
COMMITTED VERTEX NUMBER	08390h	VF	2
COMMITTED INSTANCE ID	08394h	VF	2
COMMITTED PRIMITIVE ID	08398h	VF	2
STATUS	0839Ch	VF	2
COMMON VERTEX	083A0h	VF	2
VF_GUARDBAND	083A4h	VF	2
VF REGISTERS	08490h - 084C8h	VF	30
3DSTATE_PTBR_TILE_SELECT		VF	2
NOOP		VF	1
VFL Context		VFL	496
OAR Context State		OAR	192
NOOP		LBCF	1
MI_LOAD_REGISTER_IMM	0x1100_100D	LBCF	1
LSQCREG1	0xB100	LBCF	2
LSQCREG4	0xB118	LBCF	2
LSQCREG5	0xB158	LBCF	2
LSQCREG6	0xB15C	LBCF	2
L3ALLOCREG	0xB134	LBCF	2
L3TCCNTLREG	0xB138	LBCF	2
L3RCS0RSVD1	0xB168	LBCF	2
3DSTATE_CONSTANT_VS_Commited		SVG	11
3DSTATE_CONSTANT_HS_Commited		SVG	11

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_CONSTANT_DS_Commited		SVG	11
3DSTATE_CONSTANT_GS_Commited		SVG	11
3DSTATE_VERTEX_ELEMENTS		SVG	69
3DSTATE_VF_COMPONENT_PACKING		SVG	5
3DSTATE_VF_SGVS		SVG	2
3DSTATE_VF_SGVS_2		SVG	3
3DSTATE_VS		SVG	9
3DSTATE_BINDING_TABLE_POINTERS_VS		SVG	2
3DSTATE_SAMPLER_STATE_POINTERS_VS		SVG	2
3DSTATE_URB_ALLOC_VS		SVG	3
3DSTATE_STREAMOUT		SVG	5
3DSTATE_SO_BUFFER_INDEX_0		SVG	8
3DSTATE_SO_BUFFER_INDEX_1		SVG	8
3DSTATE_SO_BUFFER_INDEX_2		SVG	8
3DSTATE_SO_BUFFER_INDEX_3		SVG	8
3DSTATE_CLIP		SVG	4
3DSTATE_PRIMITIVE_REPLICATION		SVG	6
3DSTATE_CLIP_MESH		SVG	2
3DSTATE_SF		SVG	4
3DSTATE_SCISSOR_STATE_POINTERS		SVG	2
3DSTATE_VIEWPORT_STATE_POINTERS_CL_SF		SVG	2
3DSTATE_RASTER		SVG	5
3DSTATE_TBIMR_TILE_PASS_INFO		SVG	4
3DSTATE_WM_HZ_OP		SVG	6
3DSTATE_MULTISAMPLE		SVG	2
3DSTATE_HS		SVG	9
3DSTATE_BINDING_TABLE_POINTERS_HS		SVG	2
3DSTATE_SAMPLER_STATE_POINTERS_HS		SVG	2
3DSTATE_URB_ALLOC_HS		SVG	3
3DSTATE_TASK_CONTROL		SVG	3
3DSTATE_TASK_SHADER		SVG	7
3DSTATE_TASK_SHADER_DATA		SVG	10
3DSTATE_URB_ALLOC_TASK		SVG	3
3DSTATE_TE		SVG	5
3DSTATE_TASK_REDISTRIB		SVG	2
3DSTATE_DS		SVG	11

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_BINDING_TABLE_POINTERS_DS		SVG	2
3DSTATE_SAMPLER_STATE_POINTERS_DS		SVG	2
3DSTATE_URB_ALLOC_DS		SVG	3
3DSTATE_GS		SVG	10
3DSTATE_BINDING_TABLE_POINTERS_GS		SVG	2
3DSTATE_SAMPLER_STATE_POINTERS_GS		SVG	2
3DSTATE_URB_ALLOC_GS		SVG	3
3DSTATE_MESH_CONTROL		SVG	3
3DSTATE_MESH_SHADER_DATA		SVG	10
3DSTATE_URB_ALLOC_MESH		SVG	3
3DSTATE_MESH_SHADER		SVG	8
NOOP		SVG	10
3DSTATE_DRAWING_RECTANGLE		SVG	4
NOOP		SVG	9
MI_LOAD_REGISTER_IMM	0x1100_1013	SVG	1
FF_PERF_REG	0x6b1c	SVG	2
DEREF_CTRL	0x6000	SVG	2
FF_MODE2	0x6604	SVG	2
CULLBIT1	0x6100	SVG	2
VFLSKPD	0x62A8	SVG	2
FF_MODE	0x6210	SVG	2
TBIMR_MODE	0x6200	SVG	2
NOOP		SVG	22
NOOP		SVG	4
3DSTATE_CONSTANT_PS_comitted		SVL	11
NOOP		SVL	1
3DSTATE_WM		SVL	2
3DSTATE_VIEWPORT_STATE_POINTER_CC		SVL	2
3DSTATE_CC_STATE_POINTERS		SVL	2
3DSATE_WM_SAMPLEMASK		SVL	2
3DSTATE_WM_DEPTH_STENCIL		SVL	4
3DSTATE_WM_CHROMAKEY		SVL	2
3DSTATE_DEPTH_BUFFER		SVL	10
3DSTATE_HIER_DEPTH_BUFFER		SVL	5
3DSTATE_STENCIL_BUFFER		SVL	8
3DSTATE_CLEAR_PARAMS		SVL	3

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_CPS_POINTERS		SVL	2
3DSTATE_DEPTH_CNTL_BUFFER		SVL	5
3DSTATE_DEPTH_BOUNDS		SVL	4
3DSTATE_CPSIZE_CONTROL_BUFFER		SVL	8
3DSTATE_AMFS_TEXTURE_POINTERS		SVL	11
3DSTATE_SBE		SVL	6
3DSTATE_SBE_SWIZ		SVL	11
3DSTATE_SBE_MESH		SVL	2
3DSTATE_PS		SVL	12
3DSTATE_BINDING_TABLE_POINTERS_PS		SVL	2
STATE_SAMPLER_STATE_POINTERS_PS		SVL	2
3DSTATE_BLEND_STATE_POINTERS		SVL	2
3DSTATE_PS_EXTRA		SVL	2
3DSTATE_PS_BLEND		SVL	2
3DSTATE_AMFS		SVL	1
3DSTATE_CONSTANT_TS_POINTERS		SVL	3
3DSTATE_SAMPLE_PATTERN		SVL	9
3DSTATE_SUBSLICE_HASH_TABLE		SVL	14
NOOP		SVL	13
NOOP		SVL	2
NOOP		SVL	1
MI_LOAD_REGISTER_IMM	0x1100_1021	SVL	1
Cache_Mode_0	0x7000	SVL	2
Cache_Mode_1	0x7004	SVL	2
GT_MODE	0x7008	SVL	2
OA_CULL	0x7030	SVL	2
PSS_MODE	0x7038	SVL	2
PSS_MODE2	0x703C	SVL	2
PSS_MODE3	0x7700	SVL	2
PSS_MODE4	0x7704	SVL	2
NOOP		SVL	1
NOOP		SVL	3
NOOP		WM	1
MI_LOAD_REGISTER_IMM	0x1100_1007	WM	1
WMHWCLRVAL	0x5524	WM	2
3DSTATE_POLY_STIPPLE_PATTERN		WM	33

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_AA_LINE_PARAMS		WM	3
3DSTATE_POLY_STIPPLE_OFFSET		WM	2
3DSTATE_LINE_STIPPLE		WM	3
3DSTATE_SLICE_HASH_STATE_POINTERS		WM	2
NOOP		WM	2
NOOP		WM	15
3DSTATE_MONOFILTER_SIZE		SC	2
3DSTATE_CHROMA_KEY		SC	8
NOOP		SC	22
MI_BATCH_BUFFER_END		CSEND	1
NOOP		CSEND	127

Software Interface

This chapter describes the memory-mapped registers associated with the Memory Interface, including brief descriptions of their use. Refer to each registers description and related feature for more information on each individual bit.

The registers detailed in this chapter are used across the products and are extensions to previous projects. However, slight changes may be present in some registers (i.e., for features added or removed), or some registers may be removed entirely. These changes are clearly marked within this chapter.

Commands

This section describes the commands specific to Graphics engine

State Commands

This section covers the following commands:

- **STATE_SIP command -** Subroutine Kernel based on interrupt in kernel

Command
STATE_SIP

STATE_BASE_ADDRESS

The STATE_BASE_ADDRESS command sets the base pointers for subsequent state, instruction, and media indirect object accesses by the GPE. (See Memory Access Indirection for details.)

The following commands must be reissued following any change to the base addresses:

- 3DSTATE_PIPELINE_POINTERS
- 3DSTATE_BINDING_TABLE_POINTERS
- MEDIA_STATE_POINTERS

Execution of this command causes a full pipeline flush, thus its use should be minimized for higher performance.

Command
STATE_BASE_ADDRESS
PIPELINE_SELECT

The **Pipeline Select** state is contained within the logical context.

Memory Interface Commands for Rendering Engine

Command
MI_SET_CONTEXT
MI_TOPOLOGY_FILTER

Registers

Context Save Registers

VF Instance Count Registers

VF Instance Count Register Set		
Register Type:	MMIO_VF	
Address:	08300h - 08384h	
Default Value:	0000 0000h	
Access:	RO	
Size:	1088 bits	
Description:	Set of Registers for storing the index count values. In case of preempted drawcalls, these register store index count/number per element. For the non-preempted drawcalls, the values stored are ignored upon restore. These are saved as part of render context.	
DWord	**Bits**	**Description**
0	31:0	**Index Count 0.** Index Count value for Element 0. Format: U32
1	31:0	**Index Count 1.** Index Count value for Element 1. Format: U32
...	31:0	...
33	31:0	**Index Count 33.** Index Count value for Element 33. Format: U32

Mode and Misc. Ctrl Registers

This section contains various registers for controls and modes

Controls/Modes
MI_MODE - Mode Register for Software Interface
FF_MODE - Thread Mode Register
GFX_MODE - Graphics Mode Register
GT_MODE - GT Mode Register
SAMPLER_MODE - SAMPLER Mode Register
CACHE_MODE_1 - Cache Mode Register 1
GAFS_MODE - Mode Register for GAFS
FBC_RT_BASE_ADDR_REGISTER - FBC_RT_BASE_ADDR_REGISTER
FBC_RT_BASE_ADDR_REGISTER_UPPER - FBC_RT_BASE_ADDR_REGISTER_UPPER
CACHE_MODE_SS - Cache Mode Subslice Register

L3CNTLREG - L3 Control Register

Pipelines Statistics Counter Registers

These registers keep continuous count of statistics regarding the Graphics pipeline. They are saved and restored with context but should not be changed by software except to reset them to 0 at context creation time. Write access to the statistics counter in this section must be done through MI_LOAD_REGISTER_IMM, MI_LOAD_REGISTER_MEM, or MI_LOAD_REGISTER_REG commands in ring buffer or batch buffer. These registers may be read at any time; however, to obtain a meaningful result, a pipeline flush just prior to reading the registers is necessary to synchronize the counts with the primitive stream.

Registers
IA_VERTICES_COUNT - IA Vertices Count
IA_PRIMITIVES_COUNT - Primitives Generated By VF
VS_INVOCATION_COUNT - VS Invocation Counter
HS_INVOCATION_COUNT - HS Invocation Counter
DS_INVOCATION_COUNT - DS Invocation Counter
GS_INVOCATION_COUNT - GS Invocation Counter
GS_PRIMITIVES_COUNT - GS Primitives Counter
CL_INVOCATION_COUNT - Clipper Invocation Counter
PS_INVOCATION_COUNT - PS Invocation Count
PS_INVOCATION_COUNT_SLICE0 - PS Invocation Count for Slice0
PS_INVOCATION_COUNT_SLICE1 - PS Invocation Count for Slice1
PS_INVOCATION_COUNT_SLICE2 - PS Invocation Count for Slice2
PS_INVOCATION_COUNT_SLICE4 - PS Invocation Count for Slice4
PS_INVOCATION_COUNT_SLICE5 - PS Invocation Count for Slice5
CPS_INVOCATION_COUNT - CPS Invocation Counter
PS_DEPTH_COUNT
PS_DEPTH_COUNT_SLICE0 - PS Depth Count for Slice0
PS_DEPTH_COUNT_SLICE1 - PS Depth Count for Slice1
PS_DEPTH_COUNT_SLICE2 - PS Depth Count for Slice2
PS_DEPTH_COUNT_SLICE3 - PS Depth Count for Slice3
PS_DEPTH_COUNT_SLICE4 - PS Depth Count for Slice4
PS_DEPTH_COUNT_SLICE5 - PS Depth Count for Slice5
TIMESTAMP - Reported Timestamp Count
Stream Output 0 Write Offset
Stream Output 1 Write Offset
Stream Output 2 Write Offset
Stream Output 3 Write Offset
Window Hardware Generated Clear Value

CS_CTX_TIMESTAMP- CS Context Timestamp Count:

This register provides a mechanism to obtain cumulative run time of a GPU context on HW.

CS_CTX_TIMESTAMP - CS Context Timestamp Count

The diagram below details on when CS_CTX_TIMESTAMP run time, save/restored during a GPGPU context switch flow.

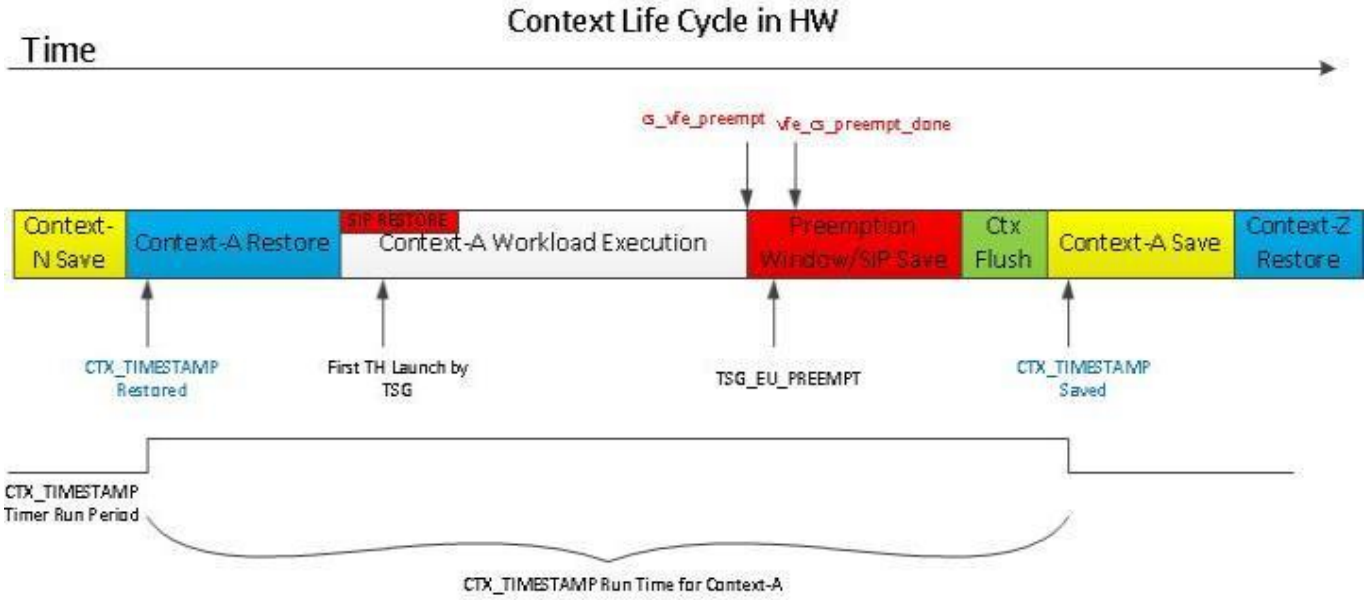


Fig: CTX_TIMESTAMP fucntionality during context execution

AUTO_DRAW Registers

AUTO_DRAW Registers
3DPRIM_END_OFFSET - Auto Draw End Offset
3DPRIM_START_VERTEX - Load Indirect Start Vertex
3DPRIM_VERTEX_COUNT - Load Indirect Vertex Count
3DPRIM_INSTANCE_COUNT - Load Indirect Instance Count
3DPRIM_START_INSTANCE - Load Indirect Start Instance
3DPRIM_BASE_VERTEX - Load Indirect Base Vertex
3DPRIM_XP0 - Load Indirect Extended Parameter 0
3DPRIM_XP1 - Load Indirect Extended Parameter 1
3DPRIM_XP2 - Load Indirect Extended Parameter 2

MMIO Registers for GPGPU Indirect Dispatch

These registers are normally written with the MI_LOAD_REGISTER_MEMORY command rather than from the CPU.

GPGPU_DISPATCHDIMX - GPGPU Dispatch Dimension X

GPGPU_DISPATCHDIMY - GPGPU Dispatch Dimension Y

GPGPU_DISPATCHDIMZ - GPGPU Dispatch Dimension Z

TS_GPGPU_THREADS_DISPATCHED - Count Active Channels Dispatched

Command Ordering Rules

There are several restrictions regarding the ordering of commands issued to the GPE. This subsection describes these restrictions along with some explanation of why they exist. Refer to the various command descriptions for additional information.

PIPELINE_SELECT

The previously active pipeline needs to be flushed immediately before switching to a different pipeline via use of the PIPELINE_SELECT command.

Refer to for details on the PIPELINE_SELECT command.

PIPELINE_SELECT

PIPE_CONTROL

The PIPE_CONTROL command does not require URB fencing/allocation to have been performed, nor does it rely on any other pipeline state. It is intended to be used on both the 3D pipe and the Media pipe. It has special optimizations to support the pipelining capability in the 3D pipe which do not apply to the Media pipe.

Common Pipeline State-Setting Commands

The following commands are used to set state common to the Graphics pipelines.

- STATE_BASE_ADDRESS
- STATE_SIP
- 3DSTATE_CHROMA_KEY
- 3DSTATE_BINDING_TABLE_POOL_ALLOC

The state variables associated with these commands must be set appropriately prior to initiating activity within a pipeline.

3D Pipeline-Specific State-Setting Commands

3D Pipeline-Specific State-Setting Commands

The following commands are used to set state specific to the 3D Pipeline.

- 3DSTATE_PIPELINED_POINTERS
- 3DSTATE_BINDING_TABLE_POINTERS
- 3DSTATE_VERTEX_BUFFERS
- 3DSTATE_VERTEX_ELEMENTS
- 3DSTATE_INDEX_BUFFERS
- 3DSTATE_VF_STATISTICS
- 3DSTATE_DRAWING_RECTANGLE
- 3DSTATE_CONSTANT_COLOR
- 3DSTATE_DEPTH_BUFFER
- 3DSTATE_POLY_STIPPLE_OFFSET
- 3DSTATE_POLY_STIPPLE_PATTERN
- 3DSTATE_LINE_STIPPLE
- 3DSTATE_GLOBAL_DEPTH_OFFSET

The state variables associated with these commands must be set appropriately prior to issuing 3DPRIMITIVE.

3DPRIMITIVE

Before issuing a 3DPRIMITIVE command, all state (with the exception of MEDIA_STATE_POINTERS) needs to be valid. Thus, the commands used to assigned that state must be issued before issuing 3DPRIMITIVE.

3D Pipeline Stages

The following table lists the various stages of the 3D pipeline and describes their major functions.

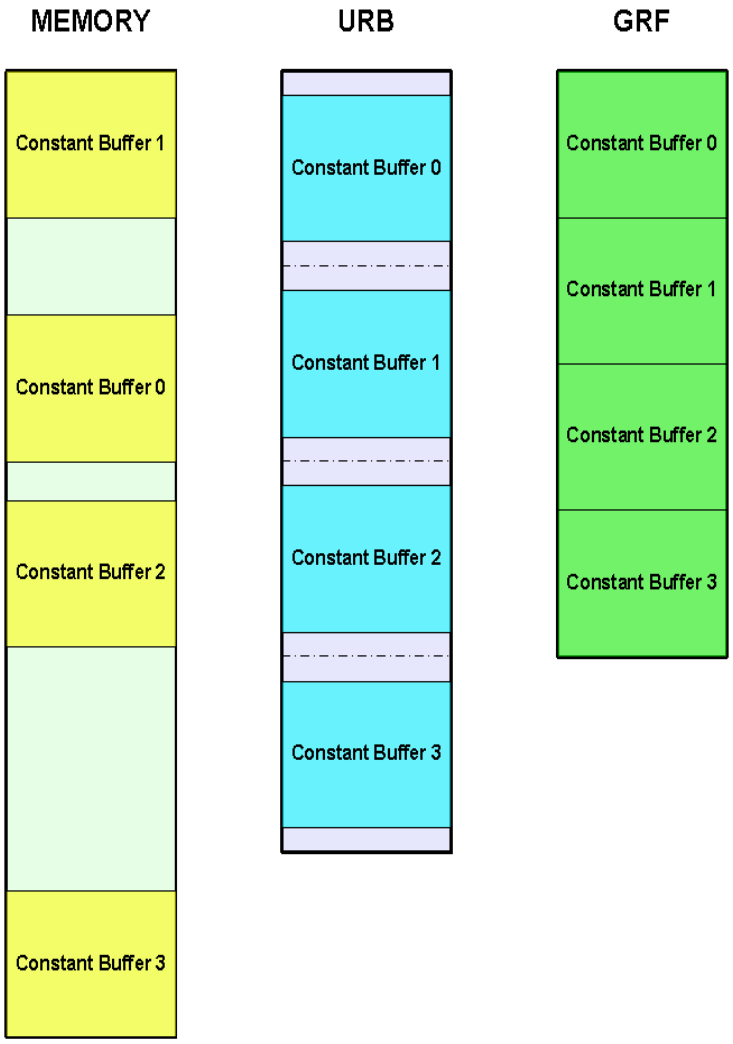
Pipeline Stage	Functions Performed
Command Stream (CS)	The Command Stream stage is responsible for managing the 3D pipeline and passing commands down the pipeline. In addition, the CS unit reads "constant data" from memory buffers and places it in the URB. Note that the CS stage is shared between the 3D, GPGPU and Media pipelines.
Vertex Fetch (VF)	The Vertex Fetch stage, in response to 3D Primitive Processing commands, is responsible for reading vertex data from memory, reformatting it, and writing the results into Vertex URB Entries. It then outputs primitives by passing references to the VUEs down the pipeline.
Vertex Shader (VS)	The Vertex Shader stage is responsible for processing (shading) incoming vertices by passing them to VS threads.
Hull Shader (HS)	The Hull Shader is responsible for processing (shading) incoming patch primitives as part of the tessellation process.
Tessellation Engine (TE)	The Tessellation Engine is responsible for using tessellation factors (computed in the HS stage) to tessellate U,V parametric domains into domain point topologies.
Domain Shader (DS)	The Domain Shader stage is responsible for processing (shading) the domain points (generated by the TE stage) into corresponding vertices.
Geometry Shader (GS)	The Geometry Shader stage is responsible for processing incoming objects by passing each object's vertices to a GS thread.
Stream Output Logic (SOL)	The Stream Output Logic is responsible for outputting incoming object vertices into Stream Out Buffers in memory.
Clipper (CLIP)	The Clipper stage performs Clip Tests on incoming objects and clips objects if required. Objects are clipped using fixed-function hardware.
Strip/Fan (SF)	The Strip/Fan stage performs object setup. Object setup uses fixed-function hardware.
Windower/Masker (WM)	The Windower/Masker performs object rasterization and determines visibility coverage.
CPS	Pipeline stage that gathers coarse pixels (CPs) for Coarse Pixel Shading (CPS).
PS Dispatch (PSD)	PSD assembles and dispatches Pixel Shader (PS) threads at one of these rates: CP, Pixel (P), or Sample (S).

3D Pipeline-Level State

This section contains table commands for the 3D Pipeline Level.

Programming Note	
Context:	3D Pipeline-Level State - Push Constant URB Allocation
The push constants are buffered in the Push Constant section of the URB. Software is required to program the hardware to allocate space in the URB for each shader push constant. The software is limited to the low addresses of the URB and must ensure that none of the shaders have overlapping handles. Software may use some if not all of the Push Constant region of the URB for pr-stage handle allocations as long as none of the push constants and handle allocations overlap. Refer to the various 3DSTATE_PUSH_CONSTANT_ALLOC_xx state commands for details regarding the maximum size of the Push Constant and other state programming information.	

Below is a diagram that represents how the hardware may move and store one CONSTANT_BUFFER command for a fixed function shader:



The bubbles in the URB are caused by the constant buffer in memory starting on a half cacheline and being an even number in length. If the constant buffer starts on an odd cacheline and has an odd number length then there will only be a bubble at the beginning of the buffer in the URB. If the constant buffer in memory starts on a cache line boundary and has an odd number length then the bubble will only be at the end of the constant buffer in the URB. Once the constant buffer is written to the GRF space then all the bubbles will be removed.

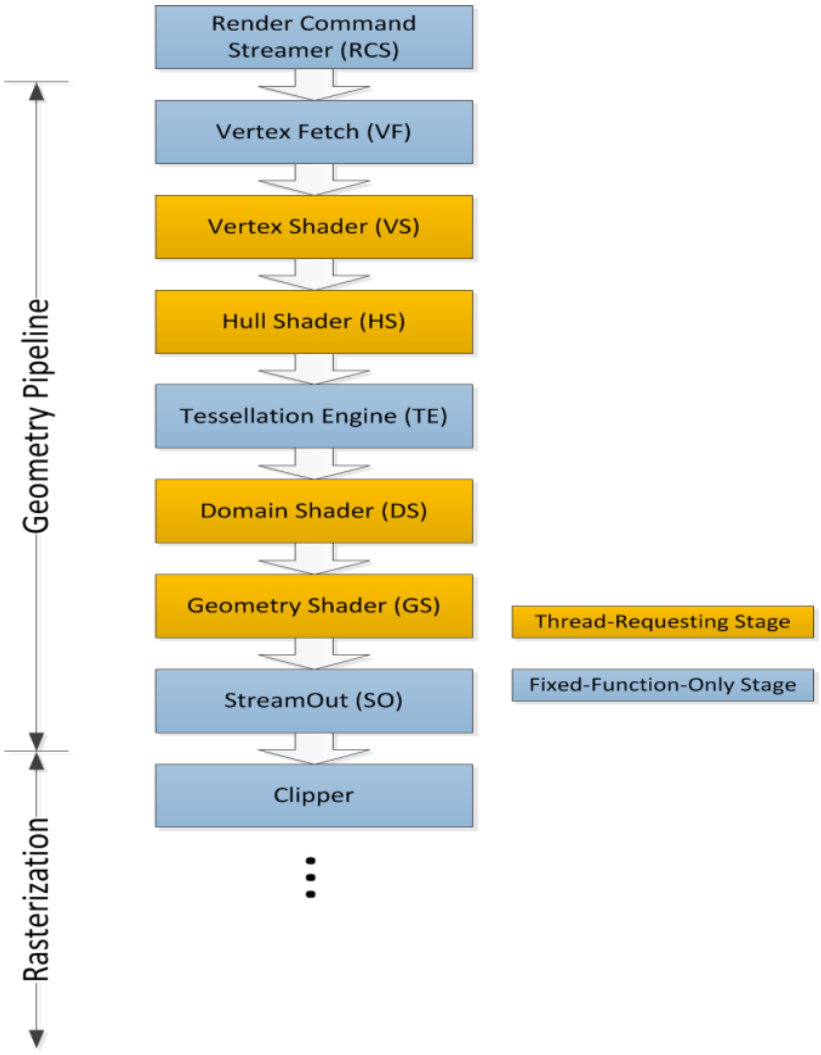
Software must guarantee that there is enough space in the push constant buffer in the URB to hold one constant buffer from memory. This includes any buffering to write the 512b aligned requests from memory into the URB.

3DSTATE_3D_MODE

3D Pipeline Geometry

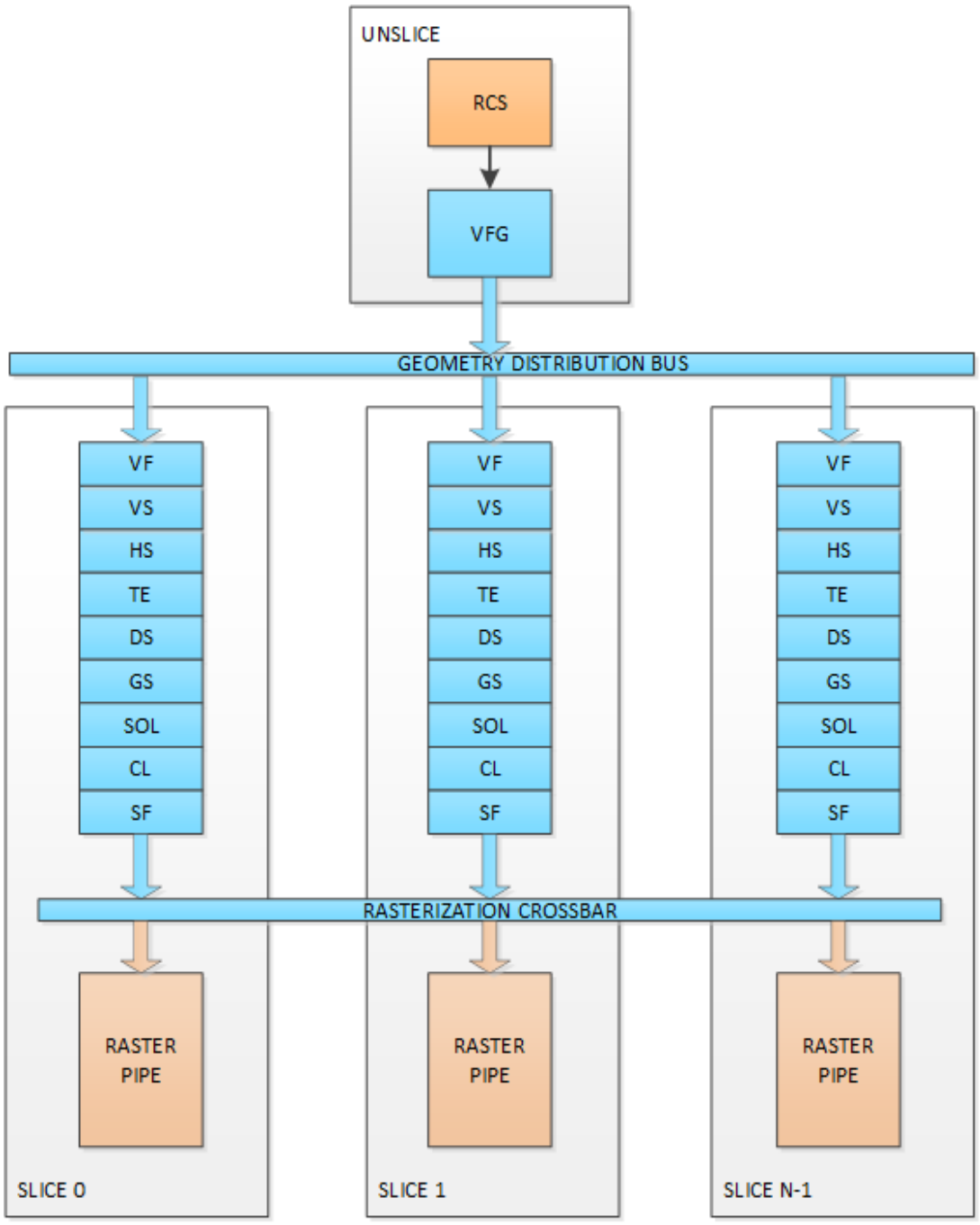
Block Diagram

The following block diagram shows the stages of the Geometry Pipeline and where they are positioned in the overall 3D Pipeline.



Multiple Geometry Pipelines

The architecture supports multiple geometry pipelines in order to provide geometry performance scaling. Each geometry pipeline is comprised of a Render pipeline. VFG stages are used to distribute work (3DPRIMITIVE commands) across the render pipelines. The RCS command streamer uses a VFG sage to distribute work across the Render pipelines, as illustrated below:



The presence of the multiple geometry pipelines is almost entirely transparent to SW. However, several state variables are available to control the work distribution process for purposes of performance tuning:

- 3DSTATE_VF:: GeometryDistributionEnable is used to switch between single and multiple geometry pipelines.
- The 3DSTATE_VFG command (described in the Vertex Fetch chapter) contains several state variables that control the granularity of which work is distributed

Mesh Shading Pipeline Overview

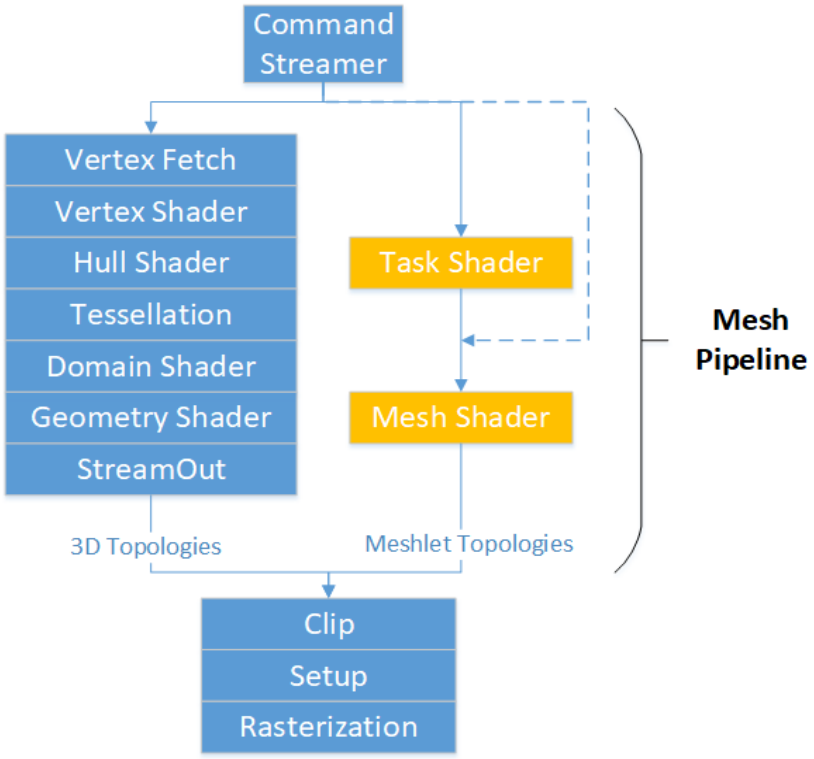
Mesh Shading is a new 3D API feature which supports the generation of geometry objects from compute-like Task and Mesh Shaders versus use of the existing 3D fixed-function geometry pipeline stages. The Mesh Shader is capable of generating "Meshlets" -- limited groupings of 3D point, line or triangle objects with both per-vertex and (new) per-primitive attributes. Meshlet generation is performed algorithmically by the Compute-like Mesh Shader kernels. An optional, preceding Task Shader can be utilized to perform high-level computation of larger 3D geometries, including high-level culling as well as subdivision of large, detailed objects into meshlet-sized groups.

The potential benefits of Mesh Shading include (but are not limited to):

- Faster Geometry Culling: FF-based culling can be a bottleneck, where performing culling in compute-like shaders allows culling TPT to scale with compute resources.
- Reduction in Draw Command counts: Task and Mesh shaders can generate a large amount of geometry from a single DrawMesh command, important in configurations with limited CPU horsepower (e.g., consoles).
- Geometry Creation Flexibility: FF Tessellation and Geometry shaders (etal) can be replaced with less restrictive and more general-purpose geometry-creation algorithms.
- Replacement for the Legacy Fixed-Function Vertex Pipeline: Though legacy app support requirements will likely continue to require FF vertex pipeline support.

The **Mesh Pipeline** appears to SW as a two-stage fixed-function pipeline which sits alongside the existing VF-SOL units and feeds into the existing Clip stage (as illustrated below). The Mesh Pipeline contains a Task Shader (TASK) and Mesh Shader (MESH) fixed-function stages.

Programming Note
While the Task and Mesh shaders logically exist in a separate pipeline, they are implemented in HW as alternate behaviors of the HS, TE and GS stages. HS performs as TaskShader, GS performs as MeshShader, and TE performs redistribution of Tasks (vs. Patches). The alternate behaviors are enabled when MeshShading is enabled, at which point the other Geometry FF stages (VF, VS, DS, and SOL) act as pass-through stages. When MeshShading in enabled, the states within those pass-through stages, as well as legacy HS, TE and DS stages, are preserved but not used until MeshShading is disabled.



While rendering is initiating in the 3D FF pipeline via the 3DPRIMITIVE command, rendering of meshlets is initiated in the Mesh pipeline via a new 3DMESH command. Execution of 3DMESH requires the Mesh Shader to be enabled, while 3DPRIMITIVE command execution requires the Mesh Shader to be disabled. The Mesh Pipeline typically performs multiple levels of visibility culling within the Task and Mesh shaders, and therefore neither requires nor supports the POSH feature.

While the 3D FF and Mesh Pipelines may operate in parallel, they share a number of resources:

- URB storage is a shared resource, typically requiring URB allocation reprogramming prior to switching between the pipelines.
- The lower 3D FF pipeline (starting with the Clip stage and including Rasterization) is a shared resource. Therefore, state reprogramming of these units may be required prior to switching pipelines. Also, output of the pipelines into the Clip stage maintains the ordering of command submission from the Command Streamer.

The Mesh Pipeline may be used with Rasterization Disabled, possibly generating outputs in memory via UAV write operations. The Mesh and Task stages also support statistics generation (counts of API-level thread dispatches).

Refer to the descriptions of the Task Shader and Mesh Shader stages for more details.

3DMESH Command Usage

The 3DMESH command is used to initiate processing in the Mesh Pipeline, analogous to the 3DPRIMITIVE command initiating processing in the 3D FF pipeline. The work initiated is a set of TaskShader threadgroup dispatch requests if the Task Shader is enabled, otherwise a set of MeshShader

threadgroup dispatch requests are initiated. The Mesh Shader must be enabled prior to submission of a 3DMESH command.

The 3DMESH command contains two parameters (possibly passed indirectly) that specify the work to be initiated:

- ThreadGroup Count
- Starting ThreadGroup ID

The 3DMESH command supports:

- Command Predication
- TBIMR
- UAV Coherency mechanisms
- Preemption, at batch-of-threadgroups granularity

General Programming of Thread-Generating Stages (VS, HS, DS, GS)

This section provides common programming information for the thread-generating Geometry FF stages (VS, HS, DS, GS). The intent is to include the common description here in order to avoid redundancy in the subsequent stage-specific sections. The stage-specific sections will include any unique or exception information, restrictions, etc. relevant to those stages.

3DSTATE_ Common State Variables

This section describes FF state variables, programmed via 3DSTATE_<FF> commands, that are common to at least two thread-generating FF stages (VS, HS, DS, GS).

The states described in these sections are only used by HW when the given stage is enabled (i.e., can request thread execution), unless specifically called out as an exception.

Thread Management State

These state variables are used by a stage to manage thread request generation.

State	VS	HS	DS	GS
Maximum Number of Threads	Y	Y	Y	Y

Maximum Number Of Threads

This field specifies, for a particular stage, the maximum number of threads allowed to be simultaneously active. Here "active" refers to (a) outstanding in the thread request queue, (b) resident in the EUs, or (c) in the thread retirement queue - up to the point the stage sees the thread retirement. Note that the sum of (a) and (c) above is non-zero, and therefore - depending on configuration - the allowed number of active threads can exceed the total number of thread slots available in the EUs.

There are two main factors to consider when programming this state variable:

- **Scratch space availability:** In the case where threads require scratch space, SW shall allocate enough contiguous scratch space for the stage to allow each active thread (as programmed by this field) to access its full per-thread allocation (as programmed by **PerThreadScratchSpace**). This may require SW to reduce **MaximumNumberOfThreads** to accommodate limitations on scratch space availability.
- **Performance:** For best performance, it is recommended that SW program this field to its maximum value. This will maximize the number of threads available to perform the stage's function. However, SW is free to program a smaller value (as long as it meets any restrictions), e.g., for performance or workaround experimentation.

Thread State Initialization State

The following values are programmed as state, subsequently included by the stage as thread request control information, and eventually loaded into an EU architectural (ARF) register upon thread dispatch. In most instances these initial values can be subsequently overwritten by the thread.

For a complete description of these EU ARF register fields, refer to the EU Execution Environment section.

These values do not appear in the thread payload. (This information may be referred to as the thread's "transparent header", as it is forwarded to the EUs but not visible in the thread payload.)

State	EU State	VS	HS	DS	GS
Kernel Start Pointer	ip[31:6]	Y	Y	Y	Y
Floating Point Mode	cr0.0[0]	Y	Y	Y	Y
Single Program Flow	cr0.0[2]	N	Y	N	Y
Vector Mask Enable	cr0.0[3]	Y	Y	Y	Y
Illegal Opcode Exception Enable	cr0.1[12]	Y	Y	Y	Y
Software Exception Enable	cr0.1[13]	Y	Y	Y	Y
Thread Priority	sr0.0[23]	Y	Y	Y	Y
Binding Table Pointer	see note	Y	Y	Y	Y

Kernel Start Pointer (KSP)

This field specifies bits [31:6] of the value loaded into the EU's **Instruction Pointer** (ip), which in turn specifies the starting offset of the kernel program to be executed. The state is specified as a 64B-granular offset from the **Instruction Base Address** register (programmed via **STATE_BASE_ADDRESS**). Bits[5:3] of the EU 'ip' register (which identify a Dword within a 64B region) are loaded with 0 upon thread dispatch.

Note (below) that **Kernel Start Pointer [47:32]** can be programmed via FF state, but these bits are ignored by HW as the EU 'ip' register only supports a 32-bit value.

A stage may support more than one KSP state, where HW performs an on-the-fly selection of one of the KSPs based on some criteria. Refer to the stage-specific sections for details. For those stages that support multiple dispatch modes but only a single KSP state, SW shall ensure that the KSP value programmed corresponds with the selected dispatch mode.

Floating Point Mode

This state bit is loaded into the EU's **Single Precision Floating Point Mode** (FPMode, cr0.0[0]) which, in turn, controls how certain single-precision floating point operations are performed within the EU subsystem.

Single Program Flow

This state bit is loaded into the EU's **Single Program Flow** (SPF, cr0.0[2]) which, in turn, controls how certain flow control instructions operate across the EU channels.

Vector Mask Enable

This state bit is loaded into the EU's **Vector Mask Enable** (VME, cr0.0[3]) which, in turn, selects whether the EU's Dispatch Mask or Vector Mask register is used as the execution mask for subsequent instructions.

Illegal Opcode Exception

This state bit is loaded into the EU's **Illegal Opcode Exception Enable** (cr0.1[12]) which, in turn, enables or disables the EU's illegal opcode exception mechanism.

Software Exception Enable

This state bit is loaded into the EU's **Software Exception Enable** (cr0.1[13]) which, in turn, enables or disables the EU's software exception mechanism.

Thread Dispatch Priority

This state bit can be used to give thread requests emanating from a Geometry FF stage higher thread dispatch priority than thread request sources that are not marked as high priority.

This state bit is also loaded into the EU's **Priority Class** (sr0.0[23]) which, in turn, determines whether the EU thread is considered as belonging to the high priority class.

Binding Table Pointer (BTP)

Upon thread request, the BTP specified for the relevant FF stage is passed to, and stored in, the EU as part of thread state. This BTP value is subsequently passed to the Shared Functions (e.g., Sampler) that are required to access surfaces specified in the Binding Table. Here the BTP is passed via a side-band channel and not directly in the message descriptor or message header.

Thread State Initialization State (Defaulted)

The following EU state variables are defaulted upon thread dispatch and therefore cannot be controlled via Geometry FF state programming. Refer to the relevant EU sections for an understanding of these state variables and whether the thread can overwrite the defaulted values. Note that this is not an exhaustive list of defaulted EU state variables, only the ones deemed most interesting for Geometry FF threads.

State	EU State	Default Value
FFID	sr0.0[27:24]	see below
Rounding Mode	cr0.0[5:4]	0
Single Precision Denorm Mode	cr0.0[7]	0
Double Precision Denorm Mode	cr0.0[6]	0
Stack Overflow Exception Enable	cr0.1[11]	0
External Halt Exception Enable	cr0.1[14]	0
Breakpoint Exception Enable	cr0.1[15]	0
Instruction Pointer [5:3]	ip[5:3]	0
Stack Pointer	sp.0	0 (see note below)
Stack Pointer Limit	sp_limit	0 (see note below)

FFID

The EU's **Fixed Function Identifier** (FFID, sr0.0[27:24]) is initialized to a value corresponding to the Geometry FF stage that requested the thread dispatch. Note that this simply identifies the source FF unit, not the specific thread dispatched.

Stack Pointer, Stack Pointer Limit

These EU state registers are defaulted to 0 for threads requested by Geometry FF units, as opposed to other thread request sources that may cause them to be initialized differently. The threads can overwrite the defaulted values if so desired.

Prefetch State

The following state variables can be used by SW to attempt the prefetch of certain state from memory into internal state cache. The prefetch is requested as part of the first thread dispatch after these state variables are specified.

Programming Restriction: Software shall not specify a prefetch region that extends into an invalid memory page, otherwise the prefetch may incur page faults.

Performance Note: Early prefetch of the state that will likely be referenced by the thread can improve thread execution performance. This is not guaranteed, especially if the amount of prefetched data is large which may result in state cache thrashing. Also, these prefetch requests are considered low priority hints by HW and may be dropped under conditions of high memory demand.

State	VS	HS	DS	GS
Sampler Count	Y	Y	Y	Y
Binding Table Entry Count	Y	Y	Y	Y

Sampler Count

This field specifies how many SAMPLER_STATE structures are prefetched from memory. The count can be specified as 0 or as a multiple of 4 (4,8,12,16). Refer to the state definition for encodings and further details.

Performance Note: It is recommended that SW program this field to (roughly) equal the number of sampler state structures referenced by the thread.

Binding Table Entry Count

This field specifies how many binding table entries (BTEs) and associated SURFACE_STATE structures are prefetched from memory. The format of this field depends on whether or not HW-generated binding tables are enabled, as determined by 3DSTATE_BINDING_TABLE_POOL_ALLOC::**BindingTablePoolEnable**.

SW Usage Note: When HW-generated binding tables are enabled, it is recommended that the Binding Table Entry Count value be generated when the shader is compiled.

HW-Generated Binding Tables Disabled:

The field has a Format of U8 and specifies a count of BTEs to be prefetched ([0,255]). Each of the SURFACE_STATE structures referenced by the BTEs will also be prefetched.

HW-Generated Binding Tables Enabled:

This field has a Format of Bitmask8 and indicates which 64B cache lines of BTEs will be fetched. Each bit in this field corresponds to a cache line, where a cache line holds 8 16-bit BTEs. Bit 0 refers to the cacheline starting at the Binding Table Pointer, as programmed by 3DSTATE_BINDING_TABLE_POINTER_xx.

By default, only the SURFACE_STATE structures referenced by the first 4 non-zero BTEs of each 64B cacheline will be prefetched.

Common Thread Payload-Related State

The following state variables are either included directly in the thread payload and/or used to control or compute other fields in the thread payload.

State	VS	HS	DS	GS
Sampler State Pointer	Y	Y	Y	Y
Scratch Space Buffer	Y	Y	Y	Y
Include Vertex Handles	N	Y	N	Y

Sampler State Pointer

This state variable specifies the starting, 32B-granular offset of the stage's SAMPLER_STATE table in memory, relative to the **DynamicStateBaseAddress**. It is programmed via 3DSTATE_SAMPLER_STATE_POINTERS_xx commands.

This value is included in thread payloads in R0.3[31:5] and is also directly propagated to the Sampler shared function for use in processing "headerless" messages. If a thread can potentially send any messages to the Sampler shared function that requires the Sampler State Pointer in the message header, that thread shall ensure that it passes along the Sampler State Pointer value passed in the thread payload.

Scratch Space

The state command specifies the Scratch Buffer memory region allocated to a stage. It is specified as a 22-bit index [27:6] into the **Surface State Base Address**. The Scratch Buffer index is passed in the thread payload in R0.5[31:10].

The surface type allocated for the Scratch Buffer shall be SURFTYPE_SCRATCH. The surface **Pitch** shall be no less than the required Per-Thread Scratch Space used by the stage. The index size of the Scratch Buffer shall be no less than the maximum number of physical threads.

Include Vertex Handles

This state variable specifies whether input vertex URB handles are included in the thread payload for threads requested by the FF stage. SW shall set this bit if the thread kernel requires access to the data contained input vertex URB entries, either in addition to or instead of the input vertex data pushed into the thread payload.

URB Payload State

The following state variables specify certain parameters related to the amount and location of URB-sourced data in the thread payload. State variables specifying other parameters are found in other state commands. Refer to the Thread Payload Overview subsection for more details.

State	VS	HS	DS	GS
Dispatch GRF Start Register for URB Data	Y	Y	Y	Y
Vertex/Patch URB Entry Read Offset	Y	Y	Y	Y
Vertex/Patch URB Entry Read Length	Y	Y	Y	Y

Dispatch GRF Start Register for URB Data

This state variable specifies a 5b GRF# (32B offset) within the thread payload where URB-sourced data starts. The URB-sourced data starts with some (possibly zero) amount of pushed Constant data, followed by some (possibly zero) amount of Vertex or Patch data.

Programming Restriction: Software shall ensure that it does not cause URB data to overwrite the R0 Header or Extended Header.

Vertex/Patch URB Entry Read Offset

This state variable specifies the 32B offset at which data is to be read from each Vertex or Patch URB entry before being included in the thread payload.

Vertex/Patch URB Entry Read Length

This state variable specifies the number of 16B (vertex elements) to be read from each Vertex or Patch URB entry, starting from the offset specified by the **Vertex/PatchURBEntryReadOffset** state.

If the read length is non-zero, SW shall ensure that the specification of the source (URB) data does not extend beyond the allocated and valid data in the URB entry. Other restrictions are described in the Thread Payload Overview subsection.

Pre-Rasterization Vertex State

The following state variables are implemented in the FF stages whose associated threads generate vertices (therefore the HS stage is excluded). The state variables control some aspects of how the generated ("output") vertices are treated if the pipeline is configured to have the stage's vertices to reach the Clip and Setup stages. Hardware determines which stage produces these "pre-rasterization" vertices as a function of which FF stages are enabled. For example, if the GS and DS stages are disabled, the VS stage's set of state variables will be used or alternatively, if the GS stage is enabled, the GS stage's set of state variables will be used.

There are "Force" state bits in the Clip & Setup stages that can be used to override use of these per-FF state variables and instead use corresponding state variables programmed in the Clip and/or Setup stages.

State	VS	HS	DS	GS
Vertex URB Entry Output Read Offset	Y	N	Y	Y
Vertex URB Entry Output Read Length	Y	N	Y	Y
User Clip Distance Clip Test Enable Bitmask	Y	N	Y	Y
User Clip Distance Cull Test Enable Bitmask	Y	N	Y	Y

Vertex URB Entry Output Read Offset

This state variable specifies the 32B offset at which attribute data is to be read from each Vertex URB entry for use by the Setup stage.

Vertex URB Entry Output Read Length

This state variable specifies the number of 16B attributes to be read from each Vertex URB entry for use by the Setup stage, starting from the offset specified by the **VertexURBEntryOutputReadOffset** state.

User Clip Distance Clip Test Enable Bitmask

This state variable is used in the Clip stage's clip test functionality. See Clip stage documentation for details.

User Clip Distance Cull Test Enable Bitmask

This state variable is used in the Clip stage's cull test functionality. See Clip stage documentation for details.

UAV Access State

This state variable is used by the HW UAV Coherency mechanism.

State	VS	HS	DS	GS	PS	Compute
Accesses UAV	Y	Y	Y	Y	Y	N

State	VS	HS	DS	GS	PS	Mesh	Task	Compute
Accesses UAV	Y	Y	Y	Y	Y	Y	Y	Y

Accesses UAV

This state bit indicates that threads requested by this FF stage may perform accesses to UAV resources. If SW enables the HW UAV Coherency function, it shall set this bit in order to include this stage in the coherency activities. For improved performance, SW should only set this bit for those FF stages that require it. If the HW UAV Coherency function is enabled, this bit is ignored.

PIPE Barrier

This state command allows for the SW to indicate to the hardware that a particular shader may require UAV coherency for any previous executed work with UAV Accesses enabled.

Accesses UAV

Statistics Enable

This state variable is used to enable/disable the statistic counter for a FF stage.

State	VS	HS	DS	GS
Statistics Enable	Y	Y	Y	Y

Statistics Enable

This state bit controls whether or not the statistic counter(s) associated with a FF stage are enabled. Refer to the specific FF stage descriptions for details on the statistics counter(s) supported.

SW shall disable statistics counting via this bit prior to submitting any 3DPRIMITIVE commands that are not to be included in statistics counting. For example, if the statistics counters are to be maintained to only track application-submitted work, SW shall ensure that any driver-generated work is not included in the statistics.

Thread State (Ignored)

The following state variables can be programmed but are ignored in the HW implementation.

State	VS	HS	DS	GS
Kernel Start Pointer [47:32]	Y	Y	Y	Y
Scratch Space Base Offset Upper	Y	Y	Y	Y

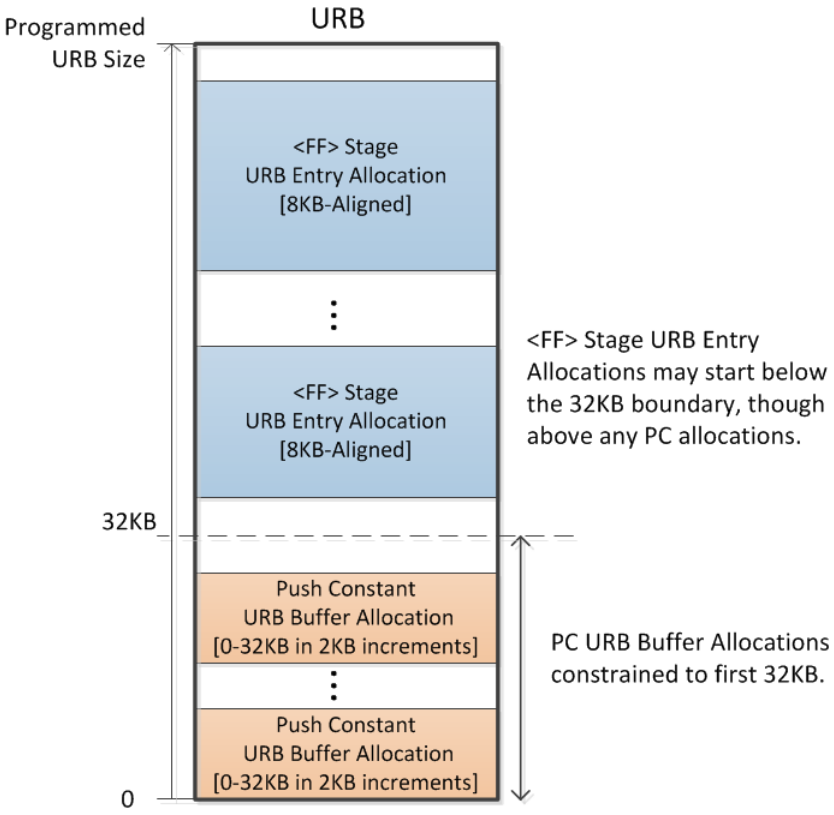
URB Allocation Overview

The Geometry FF stages use the URB for temporary storage of vertex and/or patch data as URB Entries, as well as Push Constant (PC) URB Buffers. Software can program the total size of the URB (see URB/L3 documentation). Software can also partition the URB space into FF stage-specific allocations for URB Entries and/or PC URB Buffers. These allocations can be changed dynamically to accommodate changing pipeline configurations and shader data requirements, though such changes may have performance impacts. There shall be no overlap between the individual allocations and no allocation may extend beyond the programmed URB upper limit.

Only the first 32KB of the URB can be used for VS, HS, DS, GS, and PS PC URB Buffer allocations. See *Push Constant Programming*.

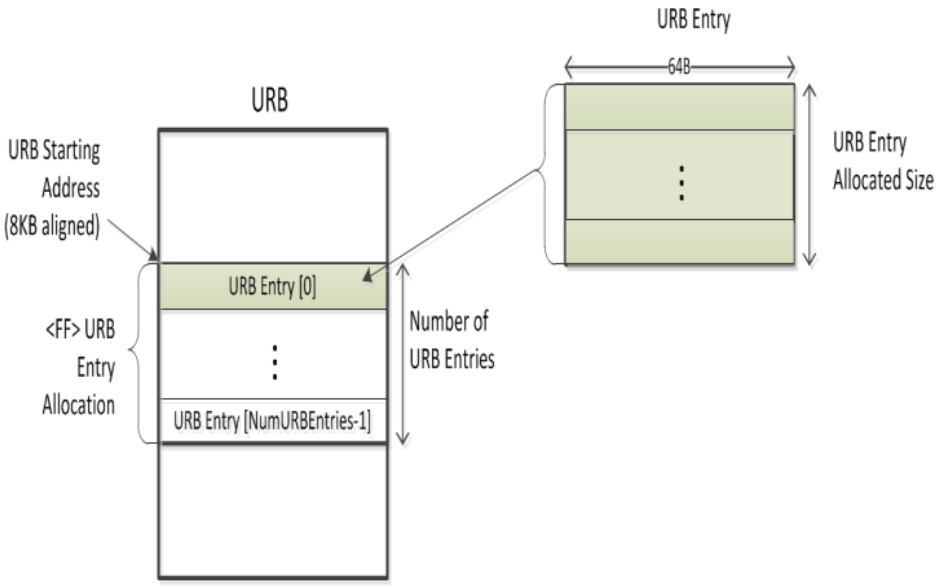
Software can place URB Entry allocations following any PC URB Buffer allocations. Software shall define allocations for all the relevant Geometry FFs (VS, HS, DS, GS), though a subset of these allocations can be "null" allocations that do not consume URB space. The VS stage always requires a non-null allocation. The HS and DS stages only require non-null allocations when tessellation is enabled. Likewise, the GS stage only requires a non-null allocation when GS is enabled.

URB Space Partitioning



The starting offset (within the URB space) of a FF URB Entry allocation is specified by a **URBStartAddress** state. The size of an allocation is defined by a **NumberOfURBEntries** state and a corresponding **URBEntryAllocationSize** state.

URB Entry and Entry Allocation



Multiple-Slice Implications

The URB programming across slices is identical. The URB programming for Slice0 shall be replicated across all slices. The "SliceN" states programmed via 3DSTATE_URB_ALLOC_* commands shall be set to the same values as the Slice0 states.

Allocation State Variables

URB Starting Address

This state variable defines the 8KB-aligned starting offset of the URB allocation for a given FF stage.

URB Entry Allocated Size

Programming Note	
Context:	Allocation State Variables

- This state variable defines the amount of storage allocated for each URB Entry within the allocation. It is 64B-granular. (Note that the contents of a URB entry can be accessed at 32B granularity). The required size of a URB Entry is typically dictated by API parameters and API shader programs.
- Software should attempt to minimize the size of URB entries in order to maximize the number of URB Entries that can be stored in a given allocation. However, as changing URB-related state variable can incur performance penalties, software may decide to employ sizing heuristics that permit a limited amount of wasted space in URB entries as a performance tradeoff.

Number of URB Entries

Programming Note	
Context:	Allocation State Variables

- This state variable defines the number of URB Entries allocated for a given FF stage. If the stage is disabled, the number of entries can be programmed as 0, though this is not required. If the stage is enabled (which is always true for the VS stage), a non-zero number of entries shall be specified. If the **URBEntryAllocatedSize** is less than or equal to 8 64B units, this number shall be 0 or a multiple of 8.
- The minimum number of entries required as well as the maximum number supported are specific to the FF stage and state programming (e.g., a function of Dispatch Mode) - see the documentation for the specific stage.

Thread Request Generation

Once a FF unit determines that a thread can be requested, it must gather all the information required to submit the thread request to the Thread Dispatcher. This information is divided into several categories, as listed below and subsequently described in detail.

- **Thread Control Information:** This is the information required (from the FF unit) to establish the execution environment of the thread.

- **Thread Payload Header:** This is the first portion of the thread payload passed in the GRF, starting at GRF R0. This is information passed directly from the FF unit. It precedes the Thread Payload Input URB Data.
- **Thread Payload Input URB Data:** This is the second portion of the thread payload. It is read from the URB using entry handles supplied by the FF unit.

Thread Control Information

The following table describes the various state variables that a FF unit uses to provide information to the Thread Dispatcher and which affect the thread execution environment. Note that this information is not directly passed to the thread in the thread payload (though some fields may be subsequently accessed by the thread via architectural registers).

State Variables Included in Thread Control Information

State Variable	Usage	FFs
Kernel Start Pointer	This field, together with the **General State Pointer**, specifies the starting location (1st EU core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Pointer**.	All FFs spawning threads
GRF Register Block Count	Specifies, in 16-register blocks, how many GRF registers are required to run the kernel. The Thread Dispatcher will only seek candidate EUs that have a sufficient number of GRF register blocks available. Upon selecting a target EU, the Thread DIspatcher will generate a logical-to-physical GRF mapping and provide this to the target EU.	All FFs spawning threads
Single Program Flow (SPF)	Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). See CR0 description in *ISA Execution Environment*.	All FFs spawning threads
Thread Dispatch Priority	The Thread Dispatcher will give priority to those thread requests with Thread Dispatch Priority of HIGH_PRIORITY over those marked as LOW_PRIORITY. Within these two classes of thread requests, the Thread Dispatcher applies a priority order (e.g., round-robin --- though this algorithm is considered a device implementation-dependent detail).	All FFs spawning threads
Floating Point Mode	This determines the initial value of the **Floating-Point Mode** bit of the EU's CR0 architectural register that controls floating point behavior in the EU core. (See ISA.)	All FFs spawning threads
Exceptions Enable	This bitmask controls the exception handing logic in the EU. (See ISA.)	All FFs spawning threads

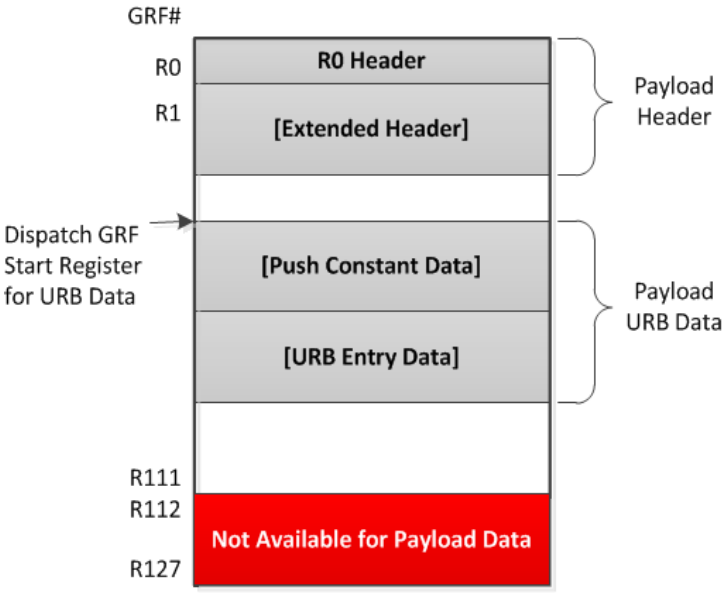
State Variable	Usage	FFs
Sampler Count	This is a hint which specifies how many indirect SAMPLER_STATE structures should be prefetched concurrent with thread initiation. It is recommended that software program this field to equal the number of samplers, though there may be some minor performance impact if this number gets large. This value should not exceed the number of samplers accessed by the thread as there would be no performance advantage. Note that the data prefetch is treated as any other memory fetch (with respect to page faults, etc.).	All stages supporting sampling (VS, GS, WM)
Binding Table Entry Count	This is a hint which specifies how many indirect BINDING_TABLE_STATE structures should be prefetched concurrent with thread initiation. (The notes included in Sampler Count (above) also apply to this field).	All FFs spawning threads

Thread Payload Overview

Like all threads, the threads spawned by Geometry FF stages have some amount of payload data pre-loaded into the GRF for use as initial input to a thread's kernel. Some of the data is sourced directly from the spawning FF and/or intermediate Thread Dispatch functions, while some is sourced from the URB as specified by the spawning FF. The Geometry FF thread payloads have a similar structure, though the exact payload size/content/layout is unique to each FF stage. This subsection describes the general layout of the payload - refer to the specific FF stage descriptions for details and differences.

The payload data loaded into the GRF starting at R0 and is divided into two main sections: the Payload Header followed by the Payload URB Data. The Payload Header contains information passed from FF units, while the Payload URB Data is obtained from the URB.

Geometry FF Thread Payload Layout (General)



The Payload Header is further subdivided into a leading R0 Header and (if present) a variable-sized Extended Header. The R0 fields are laid out to closely match the message header (M0) of thread-generated messages to shared functions. The Extended Header (if present) starts in R1 and its length varies.

The Payload URB Data section is optional and can contain a variable amount of Push Constant Data and/or a variable amount of (vertex or patch) URB Entry Data. The Payload URB Data starts at the GRF# defined by the **DispatchGRFStartRegisterForURBData** FF state variable. Software can use this state variable to place the Payload URB Data at a common starting GRF# even when the size of the Extended Header varies.

Thread Payload Generation

FF units are responsible for generating a thread *payload* - the data pre-loaded into the target EU's GRF registers (starting at R0) that serves as the primary direct input to a thread's kernel. The general format of these payloads follows a similar structure, though the exact payload size/content/layout is unique to each stage. This subsection describes the common aspects - refer to the specific stage's chapters for details on any differences.

The payload data is divided into two main sections: the *payload header* followed by the *payload URB data*. The payload header contains information passed directly from the FF unit, while the payload URB data is obtained from URB locations specified by the FF unit.

The first 256 bits of the thread payload (the initial contents of R0, aka "the R0 header") is specially formatted to closely match (and in some cases exactly match) the first 256 bits of thread-generated *messages* (i.e., the message header) accepted by shared functions. In fact, the send instruction supports having a copy of a GR's contents (such as R0) used as the message header. Software must take this intention into account (i.e., "don't muck with R0 unless you know what you're doing"). This is especially important given the fact that several fields in the R0 header are considered opaque to SW, where use or modification of their contents might lead to UNDEFINED results.

The payload header is further (loosely) divided into a leading *fixed payload header* section and a trailing, variable-sized *extended payload header* section. In general, the size, content and layout of both payload header sections are FF-specific, though many of the fixed payload header fields are common amongst the FF stages. The extended header is used by the FF unit to pass additional information specific to that FF unit. The extended header is defined to start after the fixed payload header and end at the offset defined by **Dispatch GRF Start Register for URB Data**. Software can cause use the **Dispatch GRF Start Register for URB Data** field to insert padding into the extended header in order to maintain a fixed offset for the start of the URB data.

Fixed Payload Header

The payload header is used to pass *FF pipeline information* required as thread input data. This information is a mixture of SW-provided state information (state table pointers, etc.), primitive information received by the FF unit from the FF pipeline, and parameters generated/computed by the FF unit. Most of the fields of the fixed header are common between the FF stages. These non-FF-specific

fields are described in Fixed Payload Header Fields (non-FF-specific). Note that a particular stage's header may not contain all these fields, so they are not "common" in the strictest sense.

Fixed Payload Header Fields (non-FF-specific)

Fixed Payload Header Field (non-FF-specific)	Description	FFs
FF Unit ID	Function ID of the FF unit. This value identifies the FF unit within the GPU. The FF unit uses this field (when transmitted in a Message Header to the URB Function) to detect messages emanating from its spawned threads.	All FFs spawning threads
Snapshot Flag		All FFs spawning threads
Thread ID	This field uniquely identifies this thread within the FF unit over some period.	All FFs spawning threads
Scratch Space Pointer	This is the starting location of the thread's allocated scratch space, specified as an offset from the **General State Base Address**. Note that scratch space is allocated by the FF unit on a per-thread basis, based on the **Scratch Space Base Pointer** and **Per-Thread Scratch Space Size** state variables. FF units assign a thread an arbitrarily positioned region within this space. The scratch space for multiple (API-visible) entities (vertices, pixels) is interleaved within the thread's scratch space.	All FFs spawning threads
Dispatch ID	This field identifies this thread within the outstanding threads spawned by the FF unit. This field does *not* uniquely identify the thread over any significant period. **Implementation Note:** This field is effectively an "active thread index". It is used on a thread's URB allocation request to identify which thread's handle pool is to source the allocation. It is used upon thread termination to free up the thread's scratch space allocation.	All FFs spawning threads
Binding Table Pointer	This field, together with the **Surface State Base Pointer**, specifies the starting location of the Binding Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **Surface State Base Pointer**. See *Shared Functions* for a description of a Binding Table.	All FFs spawning threads
Sampler State Pointer	This field, together with the **General State Base Pointer**, specifies the starting location of the Sampler State Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **General State Base Pointer**. See *Shared Functions* for a description of a Sampler State Table.	**All FFs spawning threads which sample (VS, GS, WM)**
Per Thread Scratch Space	This field specifies the amount of scratch space allocated to each thread spawned by the FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space.	All FFs spawning threads

Fixed Payload Header Field (non-FF-specific)	Description	FFs
Handle ID <*n*>	This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. The FF unit will use this information upon detecting a URB_WRITE message issued by the thread. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS, GS, CLIP, SF
URB Return Handle <*n*>	This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS, GS, CLIP, SF
Primitive Topology Type	As part of processing an incoming primitive, a FF unit is often required to spawn a number of threads (for example, for each individual triangle in a TRIANGLE_STRIP). This field identifies the type of primitive which is being processed by the FF unit, and which has lead to the spawning of the thread. EU kernels written to process different types of objects can use this value to direct that processing. E.g., when a CLIP kernel is to provide clipping for all the various primitive types, the kernel would need to examine the Primitive Topology Type to distinguish between point, lines, and triangle clipping requests. **Note:** In general, this field is identical to the Primitive Topology Type associated with the primitive vertices as received by the FF unit. Refer to the individual FF unit chapters for cases where the FF unit modifies the value before passing it to the thread. (for example, certain units perform toggling of TRIANGLESTRIP and TRIANGLESTRIP_REV).	GS, CLIP, SF, WM

R0 Header

The R0 header is used to pass various parameters to threads. This information contains SW-provided state information, primitive information received by the FF unit from the FF pipeline, and parameters generated/computed by the FF unit or Thread Dispatch HW.

Below is a list and description of the R0 Header fields common to all Geometry FF thread payloads. Refer to the specific payload definitions for more details and (if relevant) other FF-specific fields.

R0 Header Field	R0 Header Location
Thread ID	R0.6[23:0]
FF Thread ID (FFTID)	R0.5[9:0]
Scratch Space Buffer	R0.5[31:10]
Sampler State Pointer	R0.3[31:5]

Thread ID

This field is a sequence number that identifies this thread within all threads spawned by the relevant FF stage over some unspecified period of time.

FF Thread ID

This field is assigned by the relevant FF stage and used to identify the thread within the set of currently outstanding threads spawned by the FF unit. It shall be included in EOT messages sent by the thread as required by the relevant message header.

Scratch Space Buffer

This field specifies the scratch space region allocated to the FF stage. Each physical thread accesses its own portion of this buffer.

Sampler State Pointer

This field is a copy of the **SamplerStatePointer** state variable programmed by SW via the 3DSTATE_<FF> commands

Extended Payload Header

The extended header is of variable-size, where inclusion of a field is determined by FF unit state programming.

In order to permit the use of common kernels (thus reducing the number of kernels required), the **Dispatch GRF Start Register for URB Data** state variable is supported in all FF stages. This SV is used to place the payload URB data at a specific starting GRF register, irrespective of the size of the extended header. A kernel can therefore reference the payload URB data at fixed GRF locations, while conditionally referencing extended payload header information.

Extended Header (R1+)

In some cases, additional FF-sourced information is passed in a variable-size Extended Header, which starts at GRF R1. Some of the field definitions are common across two or more payloads and are described below. Refer to specific payload definitions for more details and (if relevant) other FF-specific fields.

Extended Header Fields
Output URB Handles
Input URB Handles
PrimitiveIDs

Output URB Handles

This set of 16-bit fields contains the 64B-aligned offsets into the URB at which a thread is to write output URB data (i.e., vertex or patch data) via URB Write messages. In the VS thread payload URB Handles are used both for input and output. Up to 8 Output URB Handles can be included in a thread payload. In some SIMD4x2 payloads, these handles are passed in the R0 Header.

Input URB Handles

This set of 16-bit fields contains the 64B-aligned offsets into the URB at which a thread can access input URB data (i.e., vertex or patch data) via URB Read messages. In the VS thread payload URB Handles are used both for input and output. Up to 256 Input URB Handles can be passed in the Extended Header.

As it is often possible for all input URB data to be pushed in the thread payload, the thread may not require Input URB Handles. As these handles may not be needed, a corresponding **IncludeVertexHandles** state bit is typically included in the FF stage's state (via 3DSTATE_<FF>). This state bit controls whether the Input URB Handles are included in the Extended Header.

PrimitiveIDs

This set of 32-bit fields contains the **PrimitiveID** values corresponding to input objects being processed by the thread. See Vertex Fetch for a description of **PrimitiveID**. As PrimitiveID may not be required as input by the thread, a corresponding **IncludePrimitiveID** state bit is typically included in the FF stage's state (via 3DSTATE_<FF>). This state bit controls whether the PrimitiveIDs are included in the Extended Header.

Payload URB Data

In each thread payload, following the payload header, is some amount of URB-sourced data required as input to the thread. This data is divided into an optional *Constant URB Entry* (CURBE), following either by a Primitive URB Entry (WM) or a number of Vertex URB Entries (VS, GS, CLIP, SF). A FF unit only knows the location of this data in the URB, and is never exposed to the contents. For each URB entry, the FF unit will supply a sequence of handles, read offsets and read lengths. The thread dispatch subsystem will read the appropriate 256-bit locations of the URB, optionally perform swizzling (VS only), and write the results into sequential GRF registers (starting at **Dispatch GRF Start Register for URB Data**).

State Variables Controlling Payload URB Data

State Variable	Usage	FFs
Dispatch GRF Start Register for URB Data	This SV identifies the starting GRF register receiving payload URB data. Software is responsible for ensuring that URB data does not overwrite the Fixed or Extended Header portions of the payload.	FFs spawning threads
Vertex URB Entry Read Offset	This SV specifies the starting offset within VUEs from which vertex data is to be read and supplied in this stage's payloads. It is specified as a 256-bit offset into any and all VUEs passed in the payload. This SV can be used to skip over leading data in VUEs that is not required by the stage's threads (e.g., skipping over the Vertex Header data at the SF stage, as that information is not required for setup calculations). Skipping over irrelevant data can only help to improve performance. Specifying a vertex data source extending beyond the end of a vertex entry is UNDEFINED.	VS, GS
Vertex URB Entry Read Length	This SV determines the amount of vertex data (starting at **Vertex URB Entry Read Offset**) to be read from each VUEs and passed into the payload URB data. It is specified in 256-bit units. A zero value is INVALID (at very least one 256-bit unit must be read). Specifying a vertex data source extending beyond the end of a VUE is UNDEFINED.	

Programming Restrictions: (others may already been mentioned)

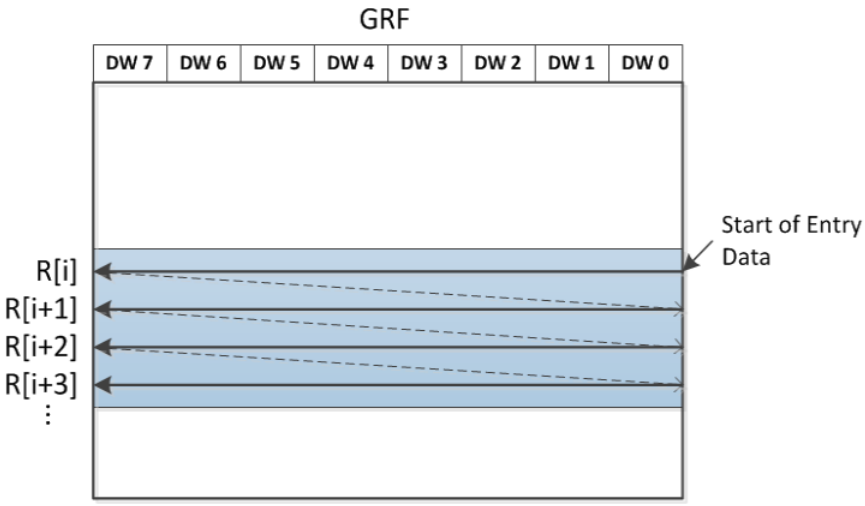
- The maximum size payload for any thread is limited by the number of GRF registers available to the thread, as determined by min (128, 16 * **GRF Register Block Count**). Software is responsible for ensuring this maximum size is not exceeded, taking into account:
  - The size of the Fixed and Extended Payload Header associated with the FF unit.
  - The **Dispatch GRF Start Register for URB** Data SV.
  - The amount of CURBE data included (via **Constant URB Entry Read Length**)
  - The number of VUEs included (as a function of FF unit, it's state programming, and incoming primitive types)
  - The amount of VUE data included for each vertex (via **Vertex URB Entry Read Length**)
  - (For WM-spawned PS threads) The amount of Primitive URB Entry data.
- For any type of URB Entry reads:
  - Specifying a source region (via Read Offset, Read Length) that goes past the end of the URB Entry allocation is illegal.
    - The allocated size of Vertex/Primitive URB Entries is determined by the **URB Entry Allocation Size** value provided in the pipeline state descriptor of the FF unit owning the VUE/PUE.
    - The allocated size of CURBE entries is determined by the **URB Entry Allocation Size** value provided in the CS_URB_STATE command.

Payload URB Data Layouts

Before going into more detail about URB-sourced payload contents, it is important to discuss the three basic layouts of this data: Linear, SIMD4x2 Interleaved, and SIMD8. These layouts are linked to how data can be accessed by the EU (therefore the EU documentation should be comprehended).

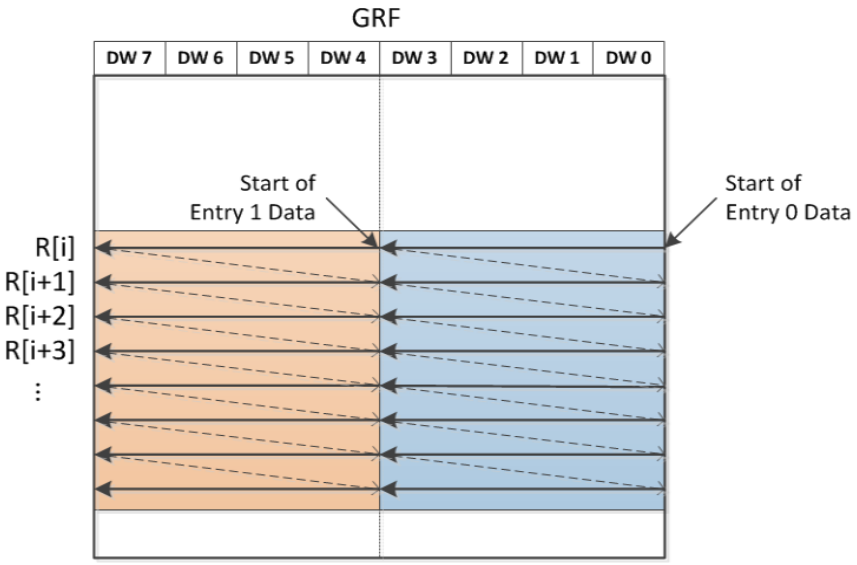
Linear

In Linear layout, data is read from the URB and placed in successive GRFs starting at DW0 of the starting destination GRF, as shown below. Data in this layout can be accessed by all EU channels of execution and it is therefore used to hold "constant" data as well as patch data for DS dispatch modes that work on a single patch at a time.



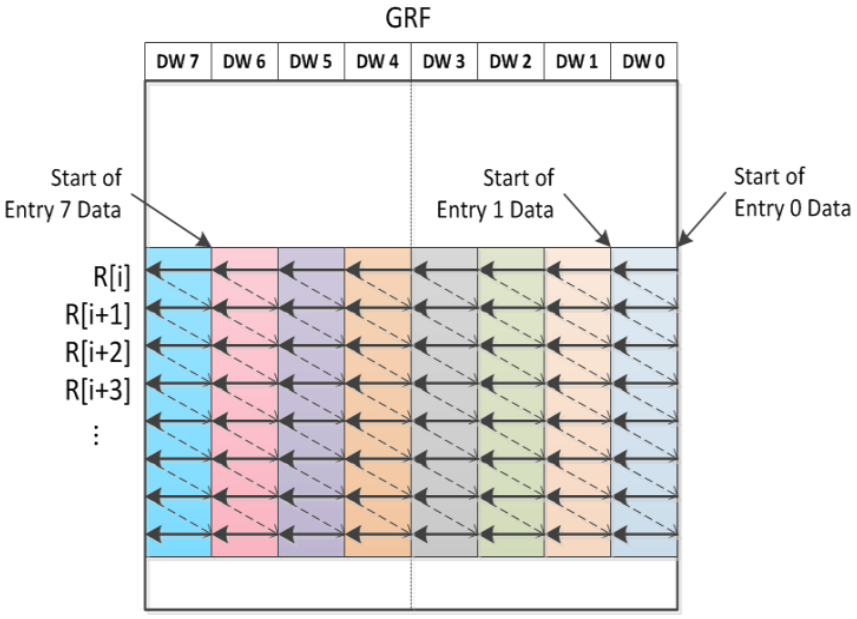
SIMD4x2 Interleaved

In SIMD4x2 Interleaved layout, the GRFs receive data from two URB entries, with the "first" URB entry loaded into the 4 lower DWs of the GRFs and the "second" URB entry loaded into the upper DWs, as shown below. This layout is primarily used to accommodate a kernel executing in SIMD4x2 execution mode (see EU documentation). It is also used to pass data from two input patches into a DS DUAL_PATCH payload, where the kernel may be executing in SIMD8 mode, but with the lower 4 SIMD8 channels operating on one patch and the upper 4 SIMD8 channels operating on another patch.



SIMD8

In SIMD8 layout, each DW position of the target GRFs can receive data from a different URB entry, as shown below. (Note that it may be possible for the data from one source URB entry to be replicated in two or more channels). This layout is used for kernels executing in SIMD8 mode, where each channel operates on independent data.



Payload URB Data

In most Geometry FF thread payloads, some amount of URB-sourced data is required as input to the thread. This data is comprised of an optional amount of Push Constant data, immediately followed by an optional amount of URB Entry data (vertex or patch data).

The starting GRF# of the Payload URB Data section is specified by the **DispatchGRFStartRegisterForURBData** per-FF state variable (programmed via 3DSTATE_<FF>). See URB Payload State above for more information on the state variables that affect the Payload URB Data.

Push Constant Data

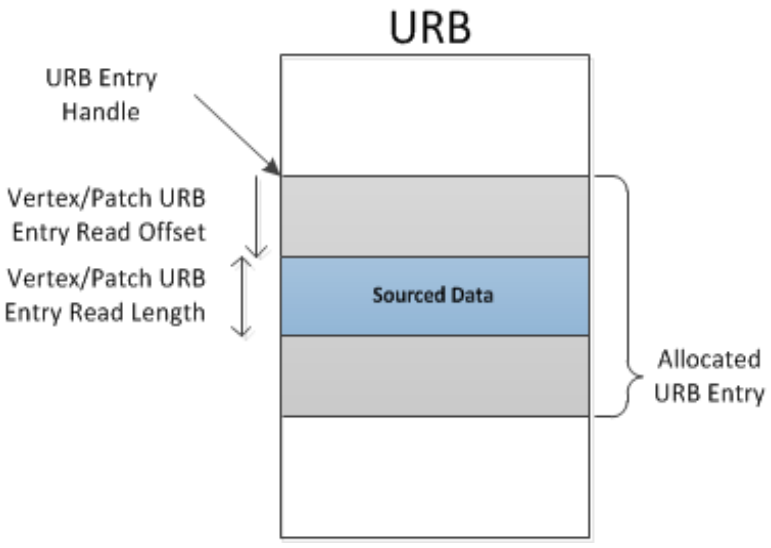
This section of the Payload URB Data is used to pass Push Constant data to the thread kernel. Software can define up to 4 Push Constant Memory Buffer regions for each Geometry FF stage that requests thread dispatches, after which the contents of those memory regions are automatically included in each subsequent payload relevant to the FF stage. A FF stage-specific Push Constant URB Buffer is used to buffer the memory contents, though any padding required by the URB buffer is removed before the data is placed in the payload. See *Push Constant Programming*.

URB Entry Data

All Geometry FF threads have some number of (vertex or patch) Input URB Entries that serve as input to the thread. Some amount (and possibly all) of the data from those Input URB Entries can be pushed into the thread payload for immediate use by the thread's kernel. While the number of Input URB Entries associated with a thread is only indirectly controlled by software (e.g., via Dispatch Mode), the source region within each of the Input URB Entries is directly programmed. This source region definition applies to all of the Input URB Entries pushed into the payload.

The diagram below shows how the **Vertex/Patch URB Entry Read Offset** and **Length** states are used to define the source region of a URB Entry that will be copied into the URB Entry Data area of the thread payload.

Input URB Entry Source Region Definition



The number of Input URB Entries pushed, and the layout of the payload data is described in the relevant FF stage descriptions.

Push Constant Programming Overview

Push constants are constant values that are pushed as part of the thread payload. Pushing constants allow for the data to be available to the Execution Units as soon as the thread payload is loaded in the GRF. The alternative to push constants is kernel-fetched constants.

All shaders (VS, HS, DS, GS and PS) have a section of the thread payload for constant data. For the geometry shaders, this is inserted between the R headers and URB Vertex Data. For Pixel Shaders, this is inserted prior to Setup Data. For more information, see the detailed descriptions of each Shader's payload in the corresponding sections.

Below is the format for the constant portion of the thread payload:

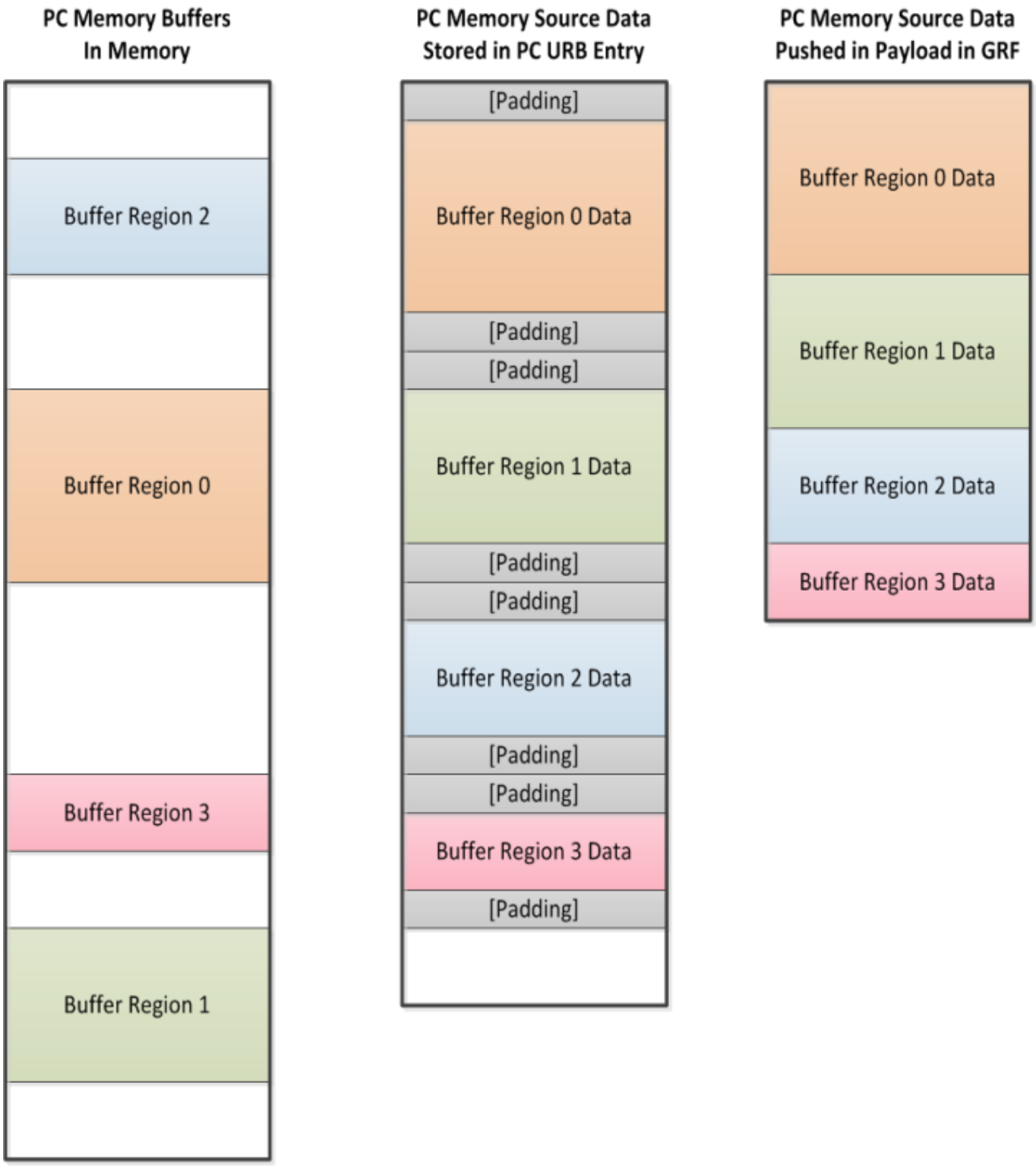
Rn		Registers prior to Push Constants
[Varies] optional	255:0	**Indirect Push Constants:** Push Constant data indirectly fetched from memory based on the 3DSTATE_CONSTANT_* command and read from the URB. The amount of data provided is defined by the sum of the read lengths in the last 3DSTATE_CONSTANT_*command
Data		Vertex or Setup Data

Indirect Push Constant Programming

Note: The Resource Streamer-based "Gather Constant" function is an extension of the Push Constant function and is described in detail in the Resource Streamer section. The Geometry FF state programming aspects are included below, after the basic Push Constant function is described.

3D APIs and their associated shader languages support the access of constant values, typically sourced from memory-resident Constant Buffers. Additionally, shader kernels may require access to compiler and/or driver-generated constants. The device supports a basic Push Constant (PC) mechanism to have a limited amount of constant data to be pushed into GRF registers via the thread payload where they are immediately available to the kernel program. It is up to software to determine which constants (if any) are pushed into the payload versus being dynamically referenced from memory via a shared function. Besides functional restrictions, there are several performance tradeoffs involved in this decision: GRF register pressure, locality of constant references, multiple references, expected shared function latency, etc.

The device supports a basic mechanism where software can specify -- for each FF stage that generates thread requests - up to 4 memory regions as the source for the PC data and one URB allocation used to buffer the data internally to the device. The device will fetch the PC source data from memory and write it into the URB Buffer, and at thread dispatch time the PC data will be read from the URB and inserted into the thread payload GRF registers.


PC Memory Buffers
In Memory
Buffer Region 2
Buffer Region 0
Buffer Region 3
Buffer Region 1
PC Memory Source Data
Stored in PC URB Entry
[Padding]
Buffer Region 0 Data
[Padding]
[Padding]
Buffer Region 1 Data
[Padding]
[Padding]
Buffer Region 2 Data
[Padding]
[Padding]
Buffer Region 3 Data
[Padding]
PC Memory Source Data
Pushed in Payload in GRF
Buffer Region 0 Data
Buffer Region 1 Data
Buffer Region 2 Data
Buffer Region 3 Data

Push Constant Memory Buffers

The 3DSTATE_CONSTANT_<FF> commands specify a set of state variables that define up to 4 PC Memory Buffer regions in memory. The commands also initiate the process of reading the PC source data (if any) from memory and placing it in the associated PC URB Buffer for inclusion in subsequent thread payloads.

Up to four PC Memory Buffers can be specified. **ConstantBufferReadLength** specifies a 32B-granular amount of PC data residing in the PC Memory Buffer. A length of 0 disables the corresponding buffer. Disabling all four buffers causes no PC data to be inserted in thread payloads. SW shall disable all four buffers whenever the corresponding PC URB Buffer is disabled. If SW disables a buffer, it shall also specify a Pointer value of 0.

The location of a PC Memory Buffer is specified either by:

- 32-Byte granular GraphicsAddress

The **GatherPoolEnable** state bit (programmed via 3DSTATE_GATHER_POOL_ALLOC) is used to enable option (c) for Buffer 1 only. Otherwise, the **CONSTANT_BUFFERAddressOffsetDisable** bit of the INSTPM register controls the use of Pointer state variables:

* If the buffers are specified via a GraphicsAddress, the **ConstantBufferObjectControlState** state variable is used to control the memory accesses, though no bounds checking is performed.

Note that the starting location and length of the PC source data in each PC Memory Buffer is specified via 32B-aligned/granular parameters, while the PC URB Buffer is specified via 64B-aligned/granular parameters. The implications of this are described in the PC URB Buffer description.

State	Command
Constant Buffer Object Control State	3DSTATE_CONSTANT_<FF>
Constant Buffer Read Length [0-3]	3DSTATE_CONSTANT_<FF>
Pointer to Constant Buffer [0-3]	3DSTATE_CONSTANT_<FF>
Pointer to Constant Buffer High [0-3]	3DSTATE_CONSTANT_<FF>

Push Constant URB Buffer Allocation

The 3DSTATE_PUSH_CONSTANT_ALLOC_<FF> commands specify a set of state variables is used to define the PC URB Buffer allocation for each relevant FF stage. Each buffer is used to collect 64B-granular/aligned PC source data prior to use in thread dispatch.

State	Command
Constant Buffer Offset	3DSTATE_PUSH_CONSTANT_ALLOC_<FF>
Constant Buffer Size	3DSTATE_PUSH_CONSTANT_ALLOC_<FF>

ConstantBufferOffset specifies the 2KB-granular offset of a FF stage's PC URB Buffer allocation. If the ConstantBufferSize is zero, this offset is ignored.

ConstantBufferSize specifies the size of a FF stage's PC URB Buffer allocation as a possibly-zero count of 2KB increments. Specifying a size of 0 disables the buffer. It is invalid to specify a non-zero amount of PC source data (via 3DSTATE_CONSTANT_<FF>) when the corresponding PC URB Buffer is disabled.

In order to use PCs for a FF stage, SW shall first program **ConstantBufferSize** to a non-zero value. The buffer shall be large enough to accommodate the worst-case buffering requirements of any single set of PC Memory Buffer definitions (see below). It is invalid to specify more PC source data than can be accommodated in the allocated PC URB Buffer. Additionally, in order to allow the device to pipeline the prefetching of subsequent PC Memory Buffers, it is recommended that SW allocate PC URB Buffers larger than this minimum requirement.

A PC URB Buffer is used to buffer 64B-granular/aligned push constant data from memory, though the PC memory regions are defined as 32B-granular/aligned. In order to accommodate the worst-case alignment, where a specific PC memory region is not 64B aligned but is 64B granular in size, the PC URB Buffer requires 32B of padding at both the beginning and end of the PC data and would therefore need to be sized at least 64B larger than the source data region wrt that source buffer. If this condition holds for all 4 PC source buffers, the PC URB Buffer needs to be sized 256B larger than the worst-case amount of source data. If SW knows a priori that the PC source data is 64B-aligned/granular, then there is no need to allocate additional room for 64B padding.

An important example of this PC URB Buffer sizing restriction is with respect to supporting a maximum amount of PC source data. The per-FF limit on the amount of PC data that can be specified for inclusion in thread payloads at any given time is 2KB (64 * 32B) spread across up to 4 source buffers. Here minimum-sized (2KB) PC URB Buffer could only be used if all the source data was 64B aligned and 64B granular is size, as the PC URB Buffer would have no room for padding. If any 64B padding was required, (at least) a 4KB PC URB Buffer would need to be allocated.

Push Constant URB Buffer Placement: SW shall program all Push Constant URB Buffer allocations to be either disabled or completely contained within the first 32KB of the URB. There are no ordering requirements on the placement of the allocations relative to the particular FF stages (e.g., the VS allocation can come before or after the GS allocation). SW shall not program enabled buffers to overlap. If 32KB is greater than the amount of URB space required for all the Push Constant URB Buffers and SW packs the allocations starting at offset 0, SW can utilize the URB space after the last allocation for URB Entry allocations (e.g., VS VUEs), subject to URB Fence alignment restrictions.

3D Primitives Overview

The 3DPRIMITIVE command (defined in the VF Stage chapter) is used to submit 3D primitives to be processed by the 3D pipeline. Typically, the processing results in the rendering of pixel data into the render targets, but this is not required.

There is considerable confusion surrounding the term 'primitive', e.g., is a triangle strip a 'primitive', or is a triangle within a triangle strip a 'primitive'? Some APIs use the term 'topology' to describe the higher-level construct (e.g., a triangle strip), and uses the term 'primitive' when discussing a triangle within a triangle strip. In this spec, we will try to avoid ambiguity by using the term 'object' to represent the basic shapes (point, line, triangle), and 'topology' to represent input geometry (strips, lists, etc.). Unfortunately, terms like '3DPRIMITIVE' must remain for legacy reasons.

The following table describes the basic primitive topology types supported in the 3D pipeline.

Programming Note	
Context:	3D Primitives Overview
• There are several variants of the basic topologies. These have been introduced to allow slight variations in behavior without requiring a state change. • Number of vertices and Dangling Vertices: Topologies have an "expected" number of vertices in order to form complete objects within the topologies (e.g., LINELIST is expected to have an even number of vertices). The actual number of vertices specified in the 3DPRIMITIVE command, and as output from the GS unit, is allowed to deviate from this expected number, in which case any "dangling" vertices are discarded. The removal of dangling vertices is initially performed in the VF unit. To filter out dangling vertices emitted by GS threads, the CLIP unit also performs dangling-vertex removal at its input.	

3D Primitive Topology Types

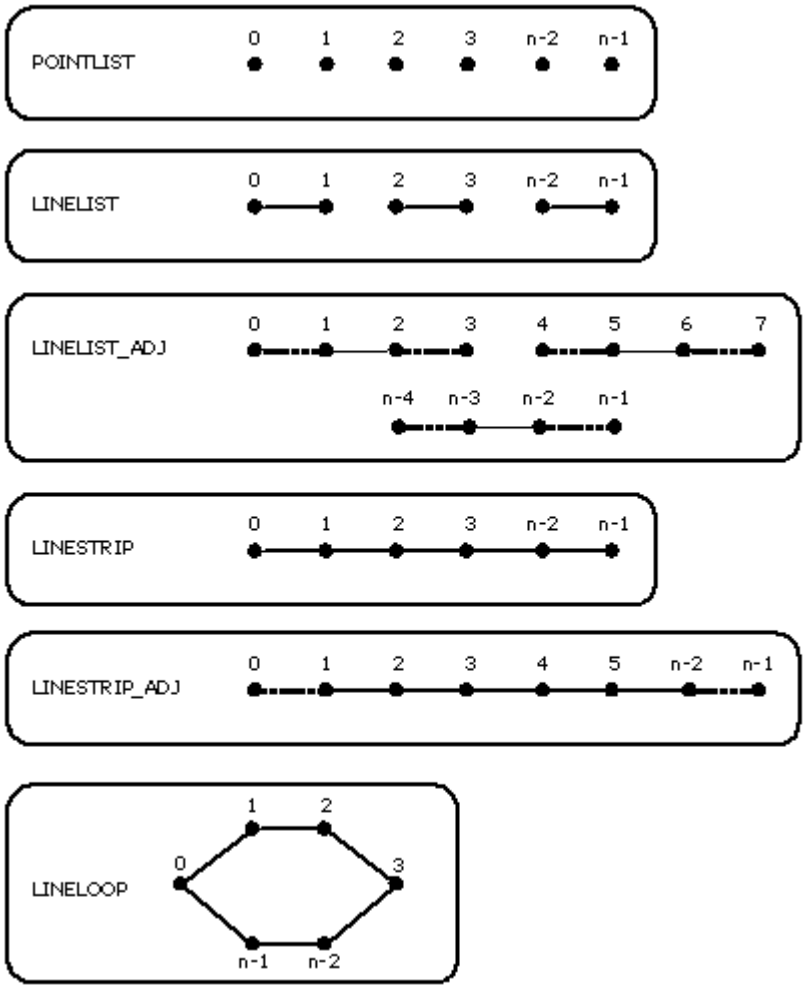
3D Primitive Topology Type (ordered alphabetically)	Description
LINELIST	• A list of independent line objects (2 vertices per line). • Normal usage expects a multiple of 2 vertices, though incomplete objects are silently ignored.
LINELIST_ADJ	• A list of independent line objects with adjacency information (4 vertices per line). • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
LINELOOP	• Similar to a 3DPRIM_LINESTRIP, though the last vertex is connected back to the initial vertex via a line object. The LINELOOP topology is converted to LINESTRIP topology at the beginning of the 3D pipeline. • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored. (The 2-vertex case is required by OGL). • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).
LINESTRIP	• A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define a connected line object. • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored.
LINESTRIP_ADJ	• A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define connected line object. The first and last segments are adjacent-only vertices. • Normal usage expects at least 4 vertices, though incomplete objects are silently

3D Primitive Topology Type (ordered alphabetically)	Description
	ignored. • Not valid as output from GS thread.
LINESTRIP_BF	• Similar to LINESTRIP, except treated as "backfacing' during rasterization (stencil test). This is used by HW to support "line" polygon fill mode when two-sided stencil is enabled. • LINESTRIP_BF is not valid as input topology.
LINESTRIP_CONT	• Similar to LINESTRIP, except LineStipple (if enabled) is continued (vs. reset) at the start of the primitive topology. • This can be used to support line stipple when the API-provided primitive is split across multiple topologies.
LINESTRIP_CONT_BF	Combination of LINESTRIP_BF and LINESTRIP_CONT variations. LINESTRIP_CONT_BF is not valid as input topology.
POINTLIST	A list of point objects (1 vertex per point).
POINTLIST_BF	• Similar to POINTLIST, except treated as "backfacing' during rasterization (stencil test). This is used to support "point" polygon fill mode when two-sided stencil is enabled. • POINTLIST_BF is not valid as input topology.
POLYGON	• Similar to TRIFAN, though the first vertex always provides the "flat-shaded" values (vs. this being programmable through state). • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
QUADLIST	• A list of independent quad objects (4 vertices per quad). • The QUADLIST topology is converted to POLYGON topology at the beginning of the 3D pipeline. • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored.
QUADSTRIP	• A list of vertices connected such that, after the first two vertices, each additional pair of vertices are associated with the previous two vertices to define a connected quad object. • Normal usage expects an even number (4 or greater) of vertices, though incomplete objects are silently ignored.
RECTLIST	• A list of independent rectangles, where only 3 vertices are provided per rectangle object, with the fourth vertex implied by the definition of a rectangle.

3D Primitive Topology Type (ordered alphabetically)	Description
	V0=LowerRight, V1=LowerLeft, V2=UpperLeft. Implied V3 = V0-V1+V2. • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored. The RECTLIST primitive is supported for 2D screen-aligned operations (e.g., BLTs and "stretch" BLTs) and not as a general 3D primitive. Due to this, a number of restrictions apply to the use of RECTLIST: • Must utilize "screen space" coordinates (VPOS_SCREENSPACE) when the primitive reaches the CLIP stage. The W component of position must be 1.0 for all vertices. The 3 vertices of each object should specify a screen-aligned rectangle (after the implied vertex is computed). • Clipping: Must not require clipping or rely on the CLIP unit's ClipTest logic to determine if clipping is required. Either the CLIP unit should be DISABLED, or the CLIP unit's Clip Mode should be set to a value other than CLIPMODE_NORMAL. • If Clipper Statistics Enable value is 2h Viewport Mapping may be ENABLED or DISABLED, if Clipper Statistics Enable value is different than 2h Viewport Mapping can be only DISABLED . If Clipper Statistics Enable value is 2h and Viewport Mapping is ENABLED, the rectlist coordinates must be in the view frustum boundaries to operate safely.
RECTLIST_SUBPIXEL	The subpixel precise, axis-aligned bounding box of the object's 3 vertices is rendered.
TRIFAN	• Triangle objects arranged in a fan (or polygon). The initial vertex is maintained as a common vertex. After the second vertex, each additional vertex is associated with the previous vertex and the common vertex to define a connected triangle object. • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRIFAN_NOSTIPPLE	• Similar to TRIFAN, but polygon stipple is not applied (even if enabled). • This can be used to support "point" polygon fill mode, under the combination of the following conditions: (a) when the frontfacing and backfacing polygon fill modes are different (so the final fill mode is not known to the driver), (b) one of the fill modes is "point" and the other is "solid", (c) point mode is being emulated by converting the point into a trifan, (d) polygon stipple is enabled. In this case, polygon stipple should not be applied to the points-emulated-as-trifans.

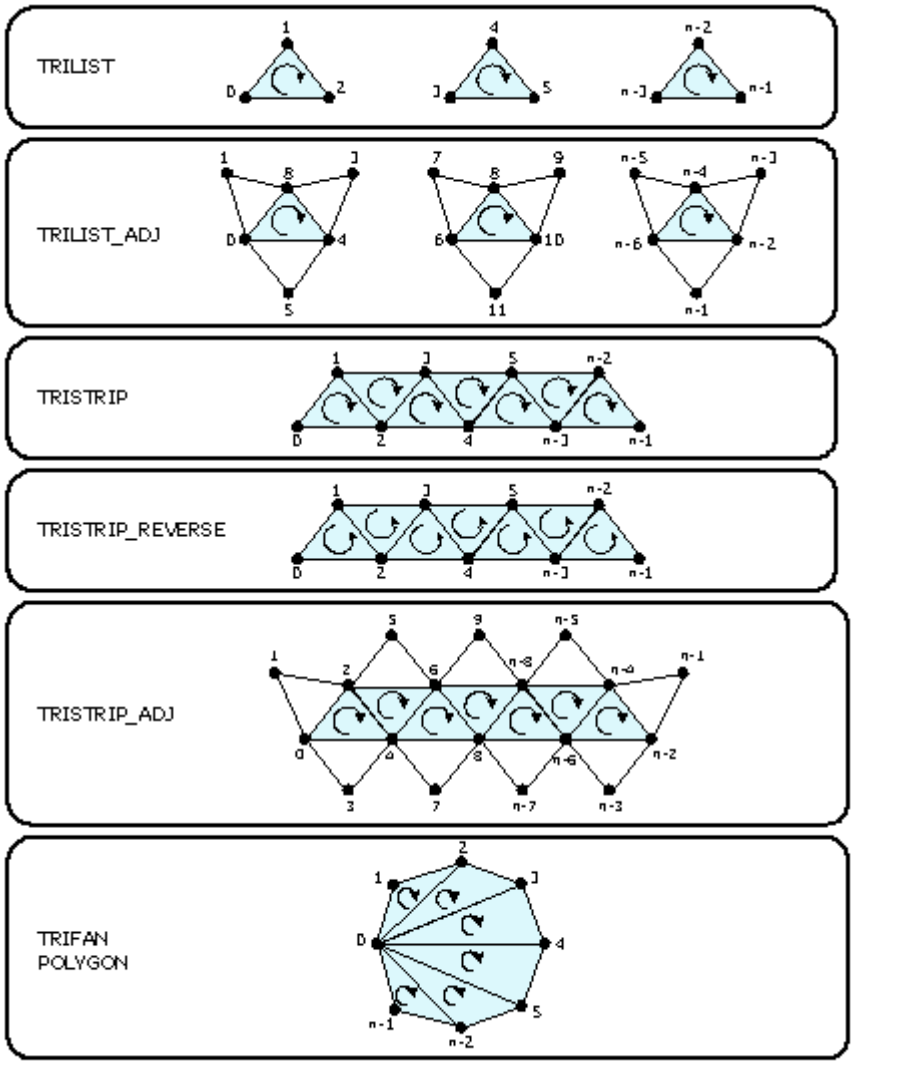
3D Primitive Topology Type (ordered alphabetically)	Description
TRILIST	• A list of independent triangle objects (3 vertices per triangle). • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.
TRILIST_ADJ	• A list of independent triangle objects with adjacency information (6 vertices per triangle). • Normal usage expects a multiple of 6 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
TRISTRIP	• A list of vertices connected such that, after the first two vertices, each additional vertex is associated with the last two vertices to define a connected triangle object. • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRISTRIP_ADJ	• A list of vertices where the even-numbered (including 0th) vertices are connected such that, after the first two vertex pairs, each additional even-numbered vertex is associated with the last two even-numbered vertices to define a connected triangle object. The odd-numbered vertices are adjacent-only vertices. • VFUNIT will complete a drawcall with the topology of tristrip_adj even if there is a preemption request in the middle of the draw call. • Normal usage expects at least 6 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
TRISTRIP_REVERSE	Similar to TRISTRIP, though the sense of orientation (winding order) is reversed - this allows SW to break long tristrips into smaller pieces and still maintain correct face orientations.
PATCHLIST_n	List of n-vertex "patch" objects. These topologies cannot be rendered directly - the tessellation units must be used to convert them into points, lines, or triangles to produce rasterization results. (VS, GS, and StreamOutput operations can also be performed.)

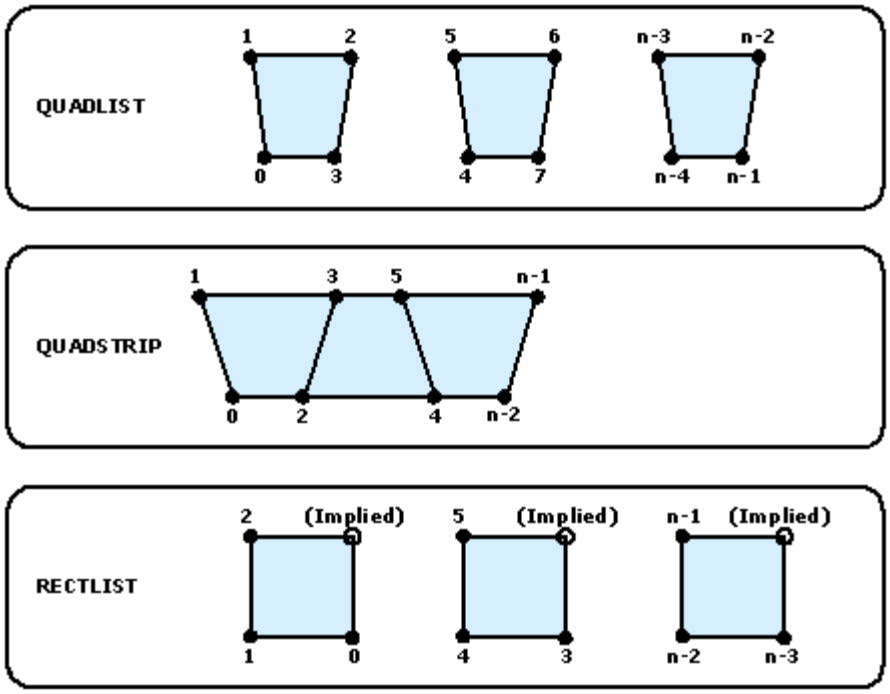
The following diagrams illustrate the basic 3D primitive topologies. (Variants are not shown if they have the same definition with respect to the information provided in the diagrams).



B6815-01

A note on the arrows you see below: These arrows are intended to show the vertex ordering of triangles that are to be considered having "clockwise" winding order in screen space. Effectively, the arrows show the order in which vertices are used in the cross-product (area, determinant) computation. Note that for TRISTRIP, this requires that either the order of odd-numbered triangles be reversed in the cross-product or the sign of the result of the normally ordered cross-product be flipped (these are identical operations).


TRILIST
TRILIST_ADJ
TRISTRIP
TRISTRIP_REVERSE
TRISTRIP_ADJ
TRIFAN
POLYGON
B6816-01



Vertex Data Overview

The information provided in this subsection refers to vertices generated by the Geometry Fixed-Function pipeline (VF through SOL stages). Most of this information is common to the Mesh Shading Pipeline that can also feed vertices to the Setup/Rasterization Pipeline. However, there are some changes (e.g., introduction of a Primitive Header) which are specific to the Mesh Shading pipeline, where the differences are described in the Mesh Shader Stage section.

The 3D pipeline FF stages (past VF) receive input 3D primitives as a stream of vertex information packets. (These packets are not directly visible to software.) Much of the data associated with a vertex is passed indirectly via a VUE handle. The information provided in vertex packets includes:

- The **URB Handle** of the VUE: This is used by the FF unit to refer to the VUE and perform any required operations on it (e.g., cause it to be read into the thread payload, dereference it, etc.).
- **Primitive Topology Information:** This information is used to identify/delineate primitive topologies in the 3D pipeline. Initially, the VF unit supplies this information, which then passes through the VS stage unchanged. GS and CLIP threads must supply this information with each vertex they produce (via the URB_WRITE message). If a FF unit directly outputs vertices (that were not generated by a thread they spawned), that FF unit is responsible for providing this information.
    - **PrimType:** The type of topology, as defined by the corresponding field of the 3DPRIMITIVE command.
    - **StartPrim:** TRUE only for the first vertex of a topology.
    - **EndPrim:** TRUE only for the last vertex of a topology.
- (Possibly, depending on FF unit) Data read back from the **Vertex Header** of the VUE.

Vertex URB Entry (VUE) Formats

In general, vertex data is stored in Vertex URB Entries (VUEs) in the URB, and only referenced by the pipeline stages indirectly via VUE handles. Therefore (for the most part) the contents/format of the vertex data is not exposed to 3D pipeline hardware - the FF units are typically only aware of the handles and sizes of VUEs.

VUEs are written in two ways:

- At the top of the 3D Geometry pipeline, the VF's InputAssembly function creates VUEs and initializes them from data extracted from Vertex Buffers as well as internally generated data.
- VS, GS, HS and DS threads can compute, format, and write new VUEs as thread output.

There are only a few points in the 3D FF pipeline where the FF units are exposed to the VUE data. Otherwise, the VUE remains opaque to the 3D pipeline hardware.

- TE stage reads back Patch Headers from Patch URB Entries
- GS stage (optionally) reads back VertexCounts and Control Data Headers from GS VUEs
- StreamOutput stage reads back VUE contents in order to stream the vertices out
- Clip stage reads back VertexHeaders from VUEs

Software must ensure that any VUEs subject to readback by the 3D pipeline start with a valid Vertex Header. This extends to all VUEs with the following exceptions:

- If the VS function is enabled, the VF-written VUEs are not required to have Vertex Headers, as the VS-incoming vertices are guaranteed to be consumed by the VS (i.e., the VS thread is responsible for overwriting the input vertex data).
- If the GS FF is enabled, neither VF-written VUEs nor VS thread-generated VUEs are required to have Vertex Headers, as the GS will consume all incoming vertices.
- If Rendering is disabled, VertexHeaders are not required anywhere.

The following table defines the Vertex Header. The Position fields are described in further detail below.

VUE Vertex Header

DWord	Bits	Description
D0	31:16	**Requested Coarse Pixel Size Y** This field specifies requested coarse pixel size in Y dimension in screen-space pixels. The value is written by the last active pre-Raster stage (VS/GS/DS). The raster stage reads this field if PER_VERTEX coarse pixel mode is set. If the last active pre-Raster stage does not write this field the value is undefined. Format: FLOAT16
	15:0	**Requested Coarse Pixel Size X** This field specifies requested coarse pixel size in X dimension in screen-space pixels. The

DWord	Bits	Description
		value is written by the last active pre-Raster stage (VS/GS/DS). The raster stage reads this field if PER_VERTEX coarse pixel mode is set. If the last active pre-Raster stage does not write this field the value is undefined. Format: FLOAT16
D1	31:0	**Render Target Array Index (RTAIndex).** This value is (eventually) used to index into a specific element of an "array" Render Target. It is read back by the GS unit (for all exiting vertices) and the Clip unit (for all clip-generated vertices), subsequently routed into the PS thread payload, and eventually included in the RTWrite DataPort message header for use by the DataPort shared function. Software is responsible for ensuring this field is zero whenever a programmable index value is not required. When a programmable index value is required , software must ensure that the correct 11-bit value is written to this field. Specifically, the kernels must perform a range check of computed index values against [0,2047], and output zero if that range is exceeded. Note that the unmodified "renderTargetArrayIndex" must be maintained in the VUE outside of the Vertex Header. Software can force an RTAIndex of 0 to be used (effectively ignoring the setting of this DWord) by use of the **ForceZeroRTAIndex** bit (3DSTATE_CLIP). Otherwise the read-back value will be used to select an RTArray element, after being clamped to the RTArray surface's [**MinimumArrayElement, Depth**] range (SURFACE_STATE). Format: 0-based U32 index value
D2	31:0	**Viewport Index.** This value is used to select one of a possible 16 sets of viewport (VP) state parameters in the Clip unit's VertexClipTest function and in the SF unit's ViewportMapping and Scissor functions. The Clip unit (if enabled) will read back this value. The Clip unit will range-check the value against [0,**Maximum VPIndex**] (see 3DSTATE_CLIP). Software can force a value of 0 to be used by programming **Maximum VPIndex** to 0. Format: 0-based U32 index value
D3	31:0	**Point Width.** This field specifies the width of POINT objects in screen-space pixels. It is used only for vertices within POINTLIST and POINTLIST_BF primitive topologies and is ignored for vertices associated with other primitive topologies. This field is read back by the Clip unit. Format: FLOAT32
D4	31:0	**Vertex Position 0 X Coordinate.** This field contains the X component of the vertex's first 4D space position. Format: FLOAT32

DWord	Bits	Description
D5	31:0	**Vertex Position 0 Y Coordinate.** This field contains the Y component of the vertex's first 4D space position. Format: FLOAT32
D6	31:0	**Vertex Position 0 Z Coordinate.** This field contains the Z component of the vertex's first NDC space position. Format: FLOAT32
D7	31:0	**Vertex Position 0 W Coordinate.** This field contains the Z component of the vertex's first 4D space position. Format: FLOAT32
If (N>0): D8 thru D[8+(4*(N-1))+3]	31:0	**Vertex Position [1].XYZW** through **Vertex Position [N].XYZW**. These fields provide, in the Render pipeline, "replica" (i.e., in addition to Vertex Position 0) 4D Vertex Positions for the vertex if the **Primitive Replication** feature is enabled via 3DSTATE_PRIMITIVE_REPLICATION. Here "**N**" refers to the value programmed in 3DSTATE_PRIMITIVE_REPLICATION::**ReplicationCount**. If N=0, the Primitive Replication feature is disabled, and therefore there shall be no additional replica Vertex Positions, i.e., only Vertex Position 0.XYWZ shall appear in the Vertex Header with the ClipDistance values (if any) immediately following Vertex Position 0. If N>0, there shall be additional replica Vertex Positions included following Vertex Position 0, and the DWord offsets of the Clip Distance Values (below, if any) shall be offset by the number of DWords required for the replica positions.
D8	31:0	**ClipDistance 0 Value (optional).** If the **UserClipDistance Clip Test Enable Bitmask** bit (3DSTATE_CLIP) is set, this value will be read from the URB in the Clip stage. If the value is found to be less than 0 or a NaN, the vertex's UCF<0> bit will set in the Clip unit's VertexClipTest function. If the **UserClipDistance Clip Test Enable Bitmask** bit is clear, this value will not be read back, and the vertex's UCF<0> bit will be zero by definition. Format: FLOAT32 ClipDistance Values are enabled for clip/cull test in the Clip stage in one of two modes: Normally the corresponding Enable Bitmasks are obtained from the state programmed in the last "vertex-producing" stage (VS/DS/GS) that is enabled prior to the Clip stage. E.g., if VS and DS are enabled but GS is disabled, the masks are obtained from 3DSTATE_DS. Alternatively, the Enable Bitmasks can be obtained directly from corresponding masks programmed via 3DSTATE_CLIP, through use of 3DSTATE_CLIP's Force User Clip Distance [Cull/Clip] Test Enable Bitmask state bits (see description of 3DSTATE_CLIP).
D9	31:0	**ClipDistance 1 Value (optional).** See above.
D10	31:0	**ClipDistance 2 Value (optional).** See above.

DWord	Bits	Description
D11	31:0	**ClipDistance 3 Value (optional).** See above.
D12	31:0	**ClipDistance 4 Value (optional).** See above.
D13	31:0	**ClipDistance 5 Value (optional).** See above.
D14	31:0	**ClipDistance 6 Value (optional).** See above.
D15	31:0	**ClipDistance 7 Value (optional).** See above.
	31:0	**End of Vertex Header Padding** (if required). The Vertex Header shall be padded at the and so that the header ends on a 32-byte boundary and therefore the Remainder of Vertex Elements (below) starts on a 32B boundary.
	31:0	**(Remainder of Vertex Elements).** The absolute maximum size limit on this data is specified via a maximum limit on the amount of data that can be read from a VUE (including the Vertex Header) (**Vertex Entry URB Read Length** has a maximum value of 63 256-bit units). Therefore, the Remainder of Vertex Elements has an absolute maximum size of 62 256-bit units. Of course, the actual allocated size of the VUE can and will limit the amount of data in a VUE.

Vertex Positions

(For brevity, the following discussion uses the term map as a shorthand for "compute screen space coordinate via perspective divide followed by viewport transform".)

The "Position" fields of the Vertex Header are the only vertex position coordinates exposed to the 3D Pipeline. The CLIP and SF units are the only FF units which perform operations using these positions. The VUE will likely contain other position attributes for the vertex outside of the Vertex Header, though this information is not directly exposed to the FF units. For example, the Clip Space position will likely be required in the VUE (outside of the Vertex Header) to perform correct and robust 3D Clipping in the CLIP thread.

CLIP unit uses the **3DSTATE_CLIP.PerspectiveDivideDisable** bit to determine whether to perform a perspective projection (divide by w) of the read-back 4D Position.

When Perspective Divide is enabled, the Clip Space position is defined in a homogeneous 4D coordinate space (pre-perspective divide), where the visible "view volume" is defined by the APIs. The API's VS, GS or DS shader program will include geometric transforms in the computation of this clip space position such that the resulting coordinate is positioned properly in relation to the view volume (i.e., it will include a "view transform" in this computation path). When Perspective Divide is enabled, the 3D FF pipeline will perform a perspective projection (division of x,y,z by w), perform clip-test on the resulting NDC (Normalized Device Coordinates), and eventually perform viewport mapping (in the SF unit) to yield screen-space (pixel) coordinates.

When Perspective Divide is disabled, the read-back Position does not undergo perspective projection by the 3D FF pipeline.

Clip Space Position

The *clip-space* position of a vertex is defined in a homogeneous 4D coordinate space where, after perspective projection (division by W), the visible "view volume" is some canonical (3D) cuboid. Typically the X/Y extents of this cuboid are [-1,+1], while the Z extents are either [-1,+1] or [0,+1]. The API's VS or GS shader program will include geometric transforms in the computation of this clip space position such that the resulting coordinate is positioned properly in relation to the view volume (i.e., it will include a "view transform" in this computation path).

Note that, under typical perspective projections, the clip-space W coordinate is equal to the view-space Z coordinate.

A vertex's clip-space coordinates must be maintained in the VUE up to 3D clipping, as this clipping is performed in clip space.

Vertex clip-space positions must be included in the Vertex Header, so that they can be read-back (prior to Clipping) and then subjected to perspective projection (in hardware) and subsequent use by the FF pipeline.

NDC Space Position

A perspective divide operation performed on a clip-space position yields a [X,Y,Z,RHW] NDC (Normalized Device Coordinates) space position. Here "normalized" means that visible geometry is located within the [-1,+1] or [0,+1] extent view volume cuboid (see clip-space above).

- The NDC X,Y,Z coordinates are the clip-space X,Y,Z coordinates (respectively) divided by the clip-space W coordinate (or, more correctly, the clip-space X,Y,Z coordinates are multiplied by the reciprocal of the clip space W coordinate).
    - Note that the X,Y,Z coordinates may contain INFINITY or NaN values (see below).
- The NDC RHW coordinate is the reciprocal of the clip-space W coordinate and therefore, under normal perspective projections, it is the reciprocal of the view-space Z coordinate. Note that NDC space is really a 3D coordinate space, where this RHW coordinate is retained in order to perform perspective-correct interpolation, etal. Note that, under typical perspective projections.
    - Note that the RHW coordinate make contain an INFINITY or NaN value (see below).

Screen-Space Position

Screen-space coordinates are defined as:

- X,Y coordinates are in absolute screen space (pixel coordinates, upper left origin). See Vertex X,Y Clamping and Quantization in the SF section for a discussion of the limitations/restrictions placed on screenspace X,Y coordinates.
- Z coordinate has been mapped into the range used for DepthTest.

- RHW coordinate is actually the reciprocal of clip-space W coordinate (typically the reciprocal of the view-space Z coordinate).

Vertex Fetch (VF) Stage

The Vertex Fetch Stage performs one major function: executing 3DPRIMITIVE commands. This is handled by the VF's InputAssembly function.

The following subsections describe some high-level concepts associated with the VF stage:

- State
- 3D Primitive Command
- Functions

Vertex Buffers as a 2D Array

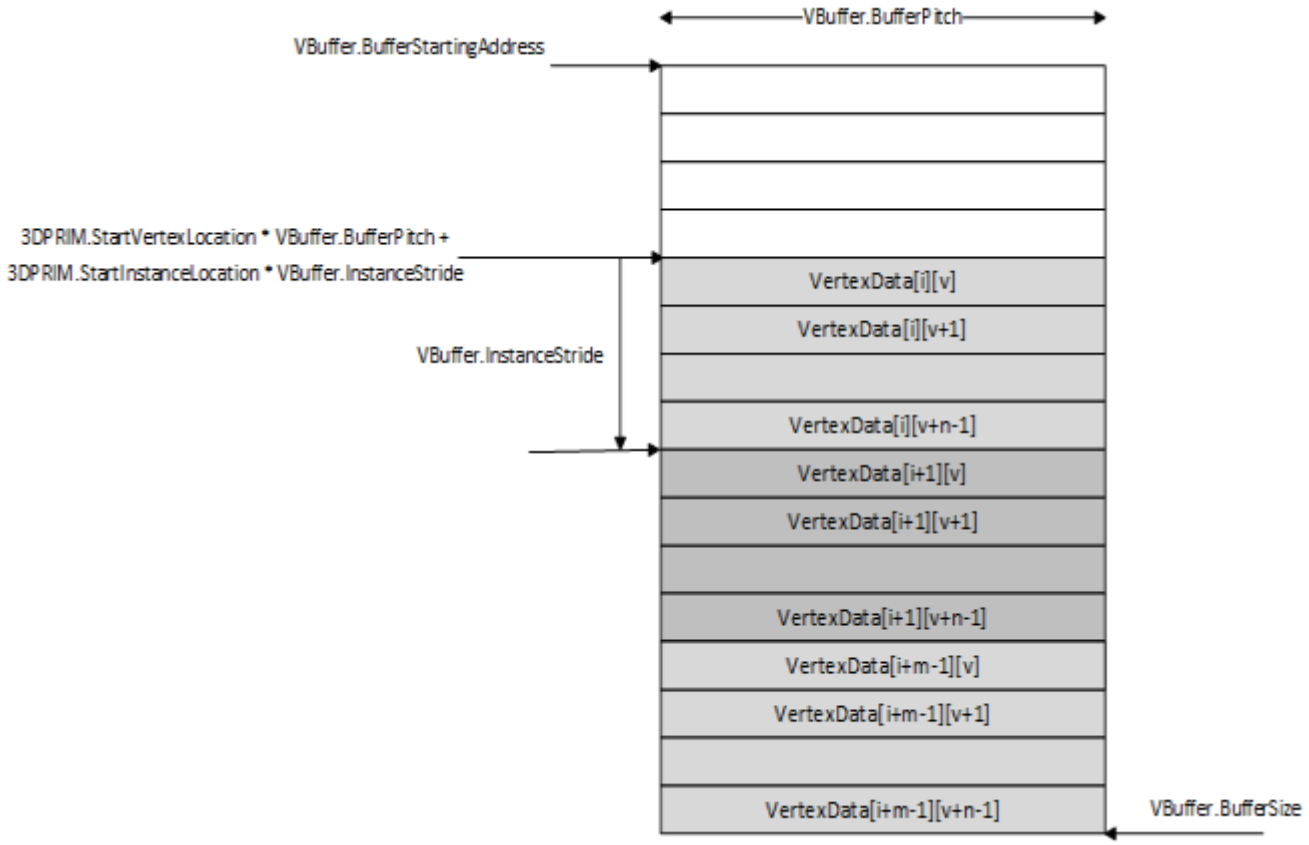
Vertex Buffers (VB) as a 2D Array

The Vertex Buffer can be accessed as a 2D array. The first dimension describes vertices where the size of the vertex is defined by the VB's **BufferPitch**. The second dimension describes instances where the size of the instance is defined by the **InstanceStride** in 3DSTATE_VF_INSTANCING.

Using the VB's as a 2D array is enabled by setting the 3DSTATE_VF_INSTANCING state **InstanceStrideEnable** to ENABLED. **InstanceStrideEnable** cannot be enabled at the same time as **InstancingEnable**.

The SGV InstanceID is used as the index for the second dimension. The state **InstanceIDOffsetEnable** can be used to provide an initial offset to InstanceID. This offset will apply to the InstanceID SGV itself and when InstanceID is used as an index for the second dimension of the VB 2D Array.

The **VertexAccessType** can be RANDOM or SEQUENTIAL when reading VB's as 2D arrays.



State

This section contains various state registers.

Control State

Register
3DSTATE_VF
3DSTATE_VF_TOPOLOGY
3DSTATE_VFG

Index Buffer (IB) State

The 3DSTATE_INDEX_BUFFER command is used to define an *Index Buffer* (IB) used in subsequent 3DPRIMITIVE commands.

The RANDOM access mode of the 3DPRIMITIVE command involves the use of a memory-resident IB. The IB, defined via the 3DSTATE_INDEX_BUFFER command described below, contains a 1D array of 8, 16 or 32-bit index values. These index values will be fetched by the InputAssembly function, and subsequently used to compute locations in VERTEXDATA buffers from which the actual vertex data is to be fetched. (This is opposed to the SEQUENTIAL access mode where the vertex data is simply fetched sequentially from the buffers).

The following table lists which primitive topology types support the presence of Cut Indices.

Definition	Cut Index?
3DPRIM_POINTLIST	Y
3DPRIM_LINELIST	Y
3DPRIM_LINESTRIP	Y
3DPRIM_TRILIST	Y
3DPRIM_TRISTRIP	Y
3DPRIM_TRIFAN	Y
3DPRIM_QUADLIST	Y
3DPRIM_QUADSTRIP	Y
3DPRIM_LINELIST_ADJ	Y
3DPRIM_LINESTRIP_ADJ	Y
3DPRIM_TRILIST_ADJ	Y
3DPRIM_TRISTRIP_ADJ	Y
3DPRIM_TRISTRIP_REVERSE	Y
3DPRIM_POLYGON	Y
3DPRIM_RECTLIST	N
3DPRIM_LINELOOP	Y
3DPRIM_POINTLIST_BF	Y
3DPRIM_LINESTRIP_CONT	Y
3DPRIM_LINESTRIP_BF	Y
3DPRIM_LINESTRIP_CONT_BF	Y
3DPRIM_TRIFAN_NOSTIPPLE	N
3DPRIM_PATCHLIST_n	Y

3DSTATE_INDEX_BUFFER

Vertex Buffers (VB) State

The 3DSTATE_VERTEX_BUFFERS and 3DSTATE_VF_INSTANCING commands are used to define *Vertex Buffers* (VBs) used in subsequent 3DPRIMITIVE commands.

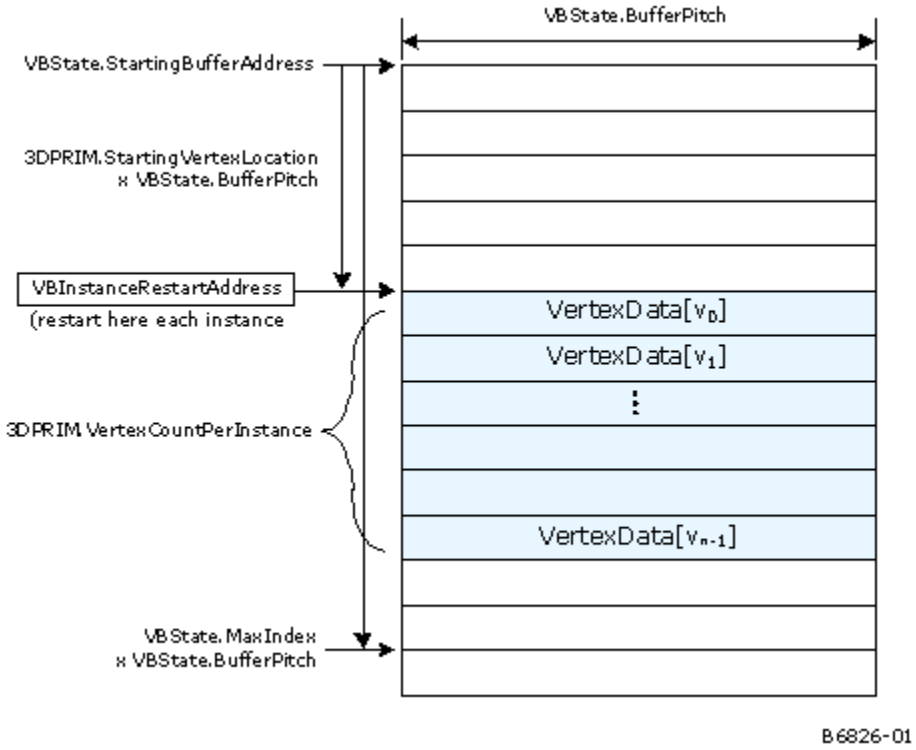
Most input vertex data is sourced from memory-resident VBs. A VB is a 1D array of structures, where the size of the structure as defined by the VB's **BufferPitch**. VBs are accessed either as *VERTEXDATA buffers* or *INSTANCEDATA buffers*, as defined by the **InstancingEnable** state in 3DSTATE_VF_INSTANCING. The VB's access type will determine whether the VF-computed VertexIndex or InstanceIndex is used to access data in the VB.

Given that the RANDOM access mode of the 3DPRIMITIVE command utilizes an IB (possibly provided by an application) to compute VB index values, VB definitions contain a **MaxIndex** value used to detect accesses beyond the end of the VBs. Any access outside the extent of a VB returns 0.

Register
3DSTATE_VERTEX_BUFFERS
VERTEX_BUFFER_STATE

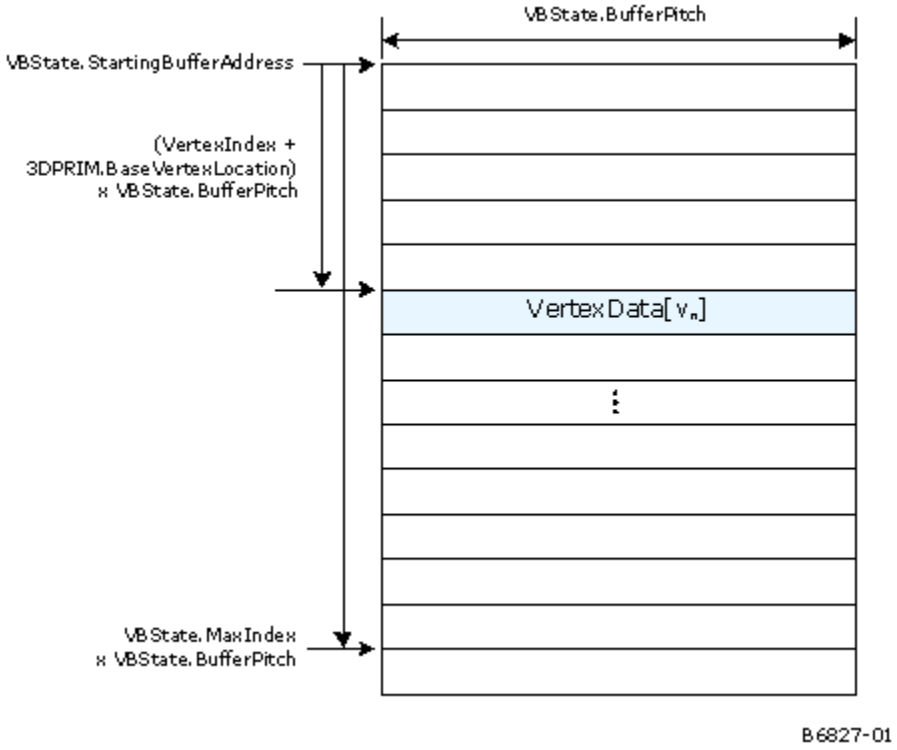
VERTEXDATA Buffers - SEQUENTIAL Access

Description
This section pertains to (a) 3DPRIMITIVE commands with VertexAccessType = SEQUENTIAL and (b) vertex elements with InstancingEnable set to DISABLED. Instead of "VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch", the address of the byte immediately beyond the last valid byte of the buffer is determined by "VBState.StartingBufferAddress + VBState.BufferSize".



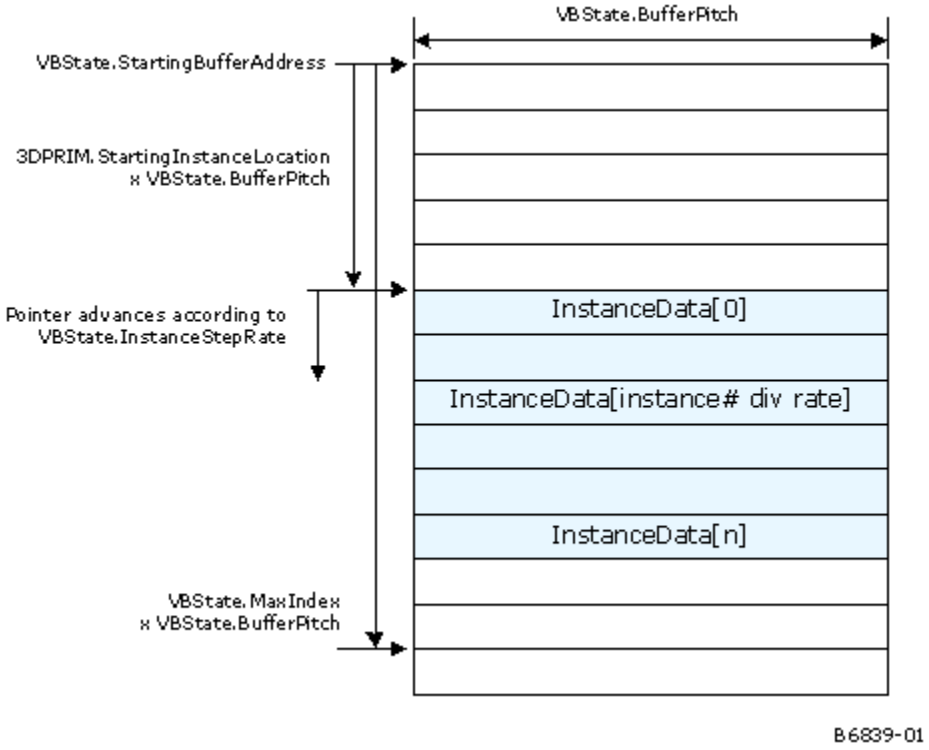
VERTEXDATA Buffers - RANDOM Access

Description
This section pertains to (a) 3DPRIMITIVE commands with VertexAccessType = RANDOM and (b) vertex elements with InstancingEnable set to DISABLED. Instead of "VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch", the address of the byte immediately beyond the last valid byte of the buffer is determined by "VBState.StartingBufferAddress + VBState.BufferSize".



INSTANCEDATA Buffers

Description
This section pertains to vertex elements with InstancingEnable set to ENABLED. Instead of "VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch", the address of the byte immediately beyond the last valid byte of the buffer is determined by "VBState.StartingBufferAddress + VBState.BufferSize".



Vertex Definition State

The following subsections define the state information for vertex data and describe some related processing.

Input Vertex Definition

The 3DSTATE_VERTEX_ELEMENTS command is used to define the source and format of input vertex data and the format of how it is stored in the destination VUE as part of 3DPRIMITIVE processing in the VF unit.

Two additional commands are added. 3DSTATE_VF_INSTANCING specifies the InstanceStepRate on a per-vertex-element basis. 3DSTATE_VF_SGVS specifies optional insertion of VertexID and/or InstanceID into the input vertex data (logically following the processing of the VERTEX_ELEMENT_STATE structures).

Refer to *3DPRIMITIVE Processing* below for the general flow of how input vertices are input and stored during processing of the 3DPRIMITIVE command.

Register
VERTEX_ELEMENT_STATE
3DSTATE_VERTEX_ELEMENTS
3D_Vertex_Component_Control
3DSTATE_VF_INSTANCING
3DSTATE_VF_SGVS
3DSTATE_VF_SGVS_2
3DSTATE_VF_COMPONENT_PACKING

3D Primitive Command

The following is the 3D Primitive Command:

3DPRIMITIVE

3D Primitive Topology Type Encoding

The following defines the encoding of the Primitive Topology Type field. See *3D Pipeline* for details, programming restrictions, diagrams, and a discussion of the basic primitive types.

3D_Prim_Topo_Type

Functions

This section covers the various functions for Vertex Fetch.

Input Assembly

The VF's InputAssembly function includes (for each vertex generated):

- Generation of VertexIndex and InstanceIndex for each vertex, possibly via use of an Index Buffer.
- Lookup of the VertexIndex in the Vertex Cache (if enabled)
- If a cache miss is detected:
  - Use of computed indices to fetch data from memory-resident vertex buffers
  - Format conversion of the fetched vertex data
  - Assembly of the format conversion results (and possibly some internally generated data) to form the complete "input" (raw) vertex
  - Storing the input vertex data in a Vertex URB Entry (VUE) in the URB
  - Output of the VUE handle of the input vertex to the VS stage
- If a cache hit is detected, the VUE handle from the Vertex Cache is passed to the VS stage (marked as a cache hit to prevent any VS processing).

Vertex Assembly

The VF utilizes a number of VERTEX_ELEMENT state structures to define the contents and format of the vertex data to be stored in Vertex URB Entries (VUEs) in the URB. See below for a detailed description of the command used to define these structures (3DSTATE_VERTEX_ELEMENTS).

Each active VERTEX_ELEMENT structure defines up to 4 contiguous DWords of VUE data, where each DWord is considered a "component" of the vertex element. The starting destination DWord offset of the vertex element in the VUE is specified, and the VERTEX_ELEMENT structures must be defined with monotonically increasing VUE offsets. For each component, the source of the component is specified. The source may be a constant (0, 0x1, or 1.0f), a generated ID (VertexID, InstanceID or PrimitiveID), or a component of a structure in memory (e.g., the Y component of an XYZW position in memory). In the case of a memory source, the Vertex Buffer sourcing the data, and the location and format of the source data with that VB are specified.

The VF's Vertex Assembly process can be envisioned as the VF unit stepping through the VERTEX_ELEMENT structures in order, fetching and format-converting the source information (if memory resident), and storing the results in the destination VUE.

The information supplied via the 3DSTATE_VF_SGVS command is also used to optionally insert VertexID and/or InstanceID into the input vertex data, after the VERTEX_ELEMENT structures are processed.

Vertex Cache

The VF stage communicates with the VS stage in order to implement a Vertex Cache function in the 3D pipeline. The Vertex Cache is strictly a performance-enhancing feature and has no impact on 3D pipeline results (other than a few statistics counters).

The Vertex Cache contains the VUE handles of VS-output (shaded) vertices if the VS function is enabled, and the VUE handles of VF-output (raw) vertices if the VS function is disabled. (Note that the actual vertex data is held in the URB, and only the handles of the vertices are stored in the cache). In either case, the contents of the cache (VUE handles) are tagged with the VertexIndex value used to fetch the input vertex data. The rationale for using the VertexIndex as the tag is that (assuming no other state or parameters change) a vertex with the same VertexIndex as a previous vertex will have the same input data, and therefore the same result from the VF+VS function.

Note that any change to the state controlling the InputAssembly function (e.g., vertex buffer definition), or any change to the state controlling the VS function (if enabled) (e.g., VS kernel), will result in the Vertex Cache being invalidated. In addition, any non-trivial use of instancing (i.e., more than one instance per 3DPRIMITIVE command and the inclusion of instance data in the input vertex) will effectively invalidate the cache between instances, as the InstanceIndex is not included in the cache tag. See Vertex Caching in *Vertex Shader* for more information on the Vertex Cache (e.g., when it is implicitly disabled, etc.)

The hardware interface to supply instancing state information is slightly different. Individual vertex elements (instead of buffers) are tagged as instanced or not.

Input Data: Push Model vs. Pull Model

Given the programmability of the pipeline, and the ability of shaders to input (load/sample) data from memory buffers in an arbitrary fashion, the decision arises in whether to push instance/vertex data into the front of the pipeline or defer the data access (pull) to the shaders that require it. Modern APIs directly support the latter model via *auto-generated IDs* in the Input Assembly function. An incrementing *VertexID*, *InstanceID*, and *PrimitiveID* are generated in the Input Assembly process, and these values can be declared as input to the "first enabled, relevant" shader. That shader can, for example, use the HW-generated ID as an index into a memory resource such as a constant buffer or vertex buffer. The 3D pipeline HW supports these IDs as required by the APIs.

There are tradeoffs involved in deciding between these models. For vertex data, it is probably always better to push the data into the pipeline, as the VF hardware attempts to cover the latency of the data fetch. The decision is less clear for instance data, as pushing instance data leads to larger Vertex URB entries which will be holding redundant data (as the instance data for vertices of an object are by definition the same). Regardless, the 3D pipeline supports both models.

Generated IDs

Note that the generated IDs are considered separate from any offset computations performed by the VF unit and are therefore described separately here.

The VF generates InstanceID, VertexID, and PrimitiveID values as part of the InputAssembly process.

VertexID and InstanceID are only allowed to be inserted into the input vertex data as it is gathered and written into the URB as a VUE.

The definition/use of PrimitiveID is more complicated than the other auto-generated IDs. PrimitiveID is associated with an "object" and not a particular vertex.

It is only available to the GS and HS as a special non-vertex input and the PS as a constant-interpolated attribute. It is not seen by the VS or DS at all.

The PrimitiveID therefore is kept separate from the vertex data. Take for example a TRILIST primitive topology: It should be possible to share vertices between triangles in the list (i.e., reuse the VS output of a vertex), even though each triangle has a different PrimitiveID associated with it.

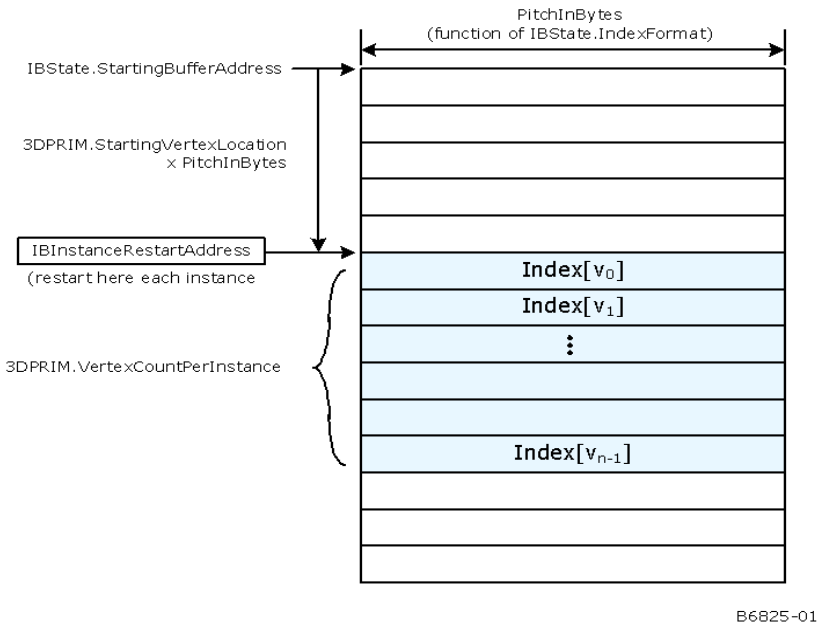
The optional insertion of VertexID and/or InstanceID into the input vertex data occurs as a separate step after the processing of VERTEX_ELEMENT structures and is controlled via the 3DSTATE_VF_SGVS command.

PrimitiveID is generated by hardware, plumbed down into the HS, GS and SF stages. It is passed along in HS/GS thread payloads. Software can also select PrimitiveID to be swizzled into vertex attribute data in the SF stage, though only if neither the HS nor GS stages are enabled.

3D Primitive Processing

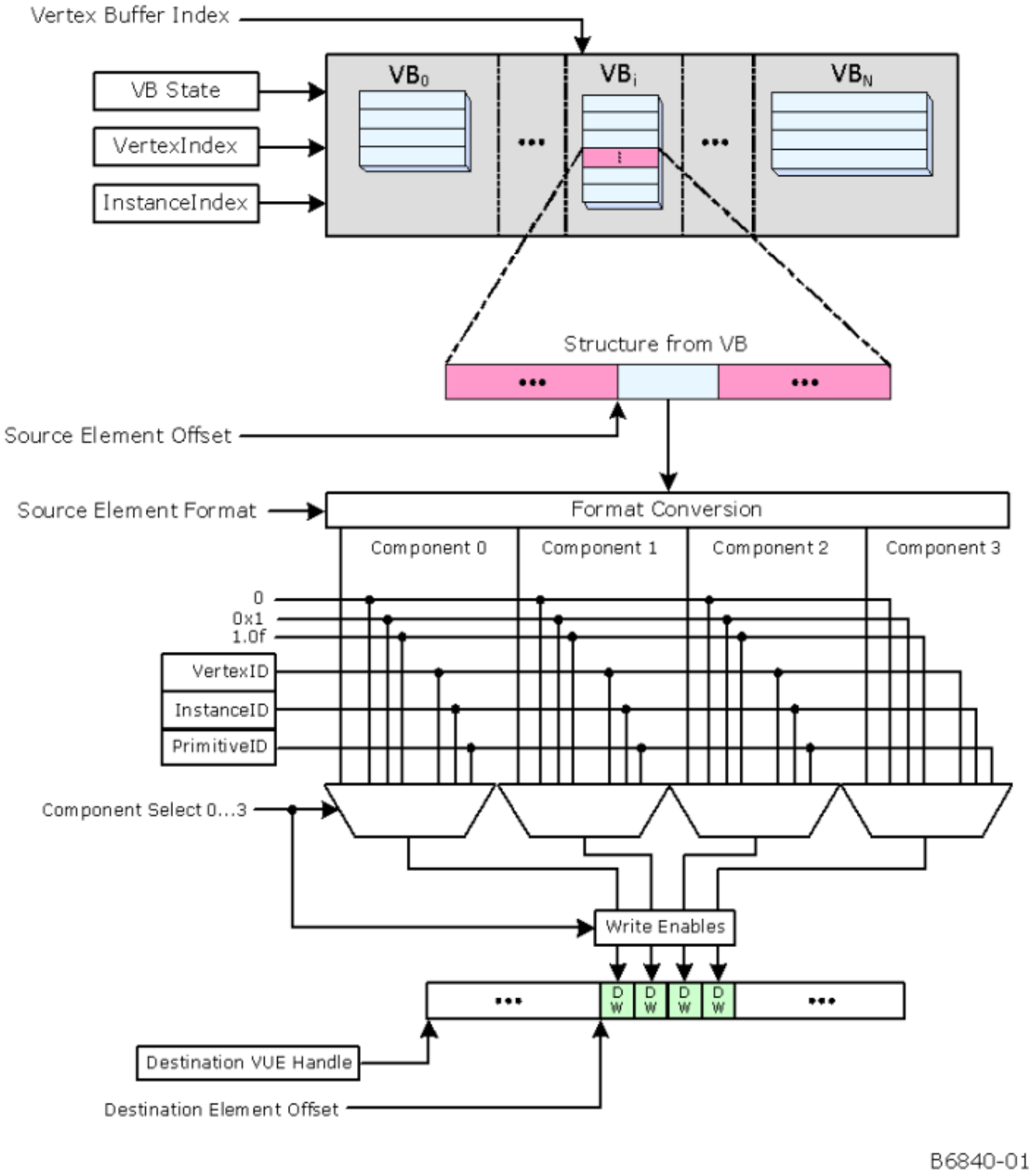
Index Buffer Access

The following figure illustrates how the Index Buffer is accessed.



Vertex Element Data Path

The following diagram shows the path by which a vertex element within the destination VUE is generated and how the fields of the VERTEX_ELEMENT_STATE structure is used to control the generation.



FormatConversion

Once the VE source data has been fetched, it is subjected to format conversion. The output of format conversion is up to 4 32-bit components, each either integer or floating-point (as specified by the **Source Element Format**). See *Sampler* for conversion algorithms.

The following table lists the valid **Source Element Format** selections, along with the format and availability of the converted components (if a component is listed as -, it cannot be used as the source of a VUE component). **Note:** This table is a subset of the list of supported surface formats defined in the *Sampler* chapter. Please refer to that table as the "original list". This table is here only to identify the components available (per format) and their format.

Source Element Formats Supported in VF Unit

Source Element	Converted Component				
Surface Format Name	Format	0	1	2	3
R32G32B32A32_FLOAT	FLOAT	R	G	B	A
R32G32B32A32_SINT	SINT	R	G	B	A
R32G32B32A32_UINT	UINT	R	G	B	A
R32G32B32A32_UNORM	FLOAT	R	G	B	A
R32G32B32A32_SNORM	FLOAT	R	G	B	A
R64G64_FLOAT	FLOAT	R	G	-	-
R32G32B32A32_SSCALED	FLOAT	R	G	B	A
R32G32B32A32_USCALED	FLOAT	R	G	B	A
R32G32B32A32_SFIXED	FLOAT	R	G	B	A
R64G64_PASSTHRU	NONE	R	G	-	-
R32G32B32_FLOAT	FLOAT	R	G	B	-
R32G32B32_SINT	SINT	R	G	B	-
R32G32B32_UINT	UINT	R	G	B	-
R32G32B32_UNORM	FLOAT	R	G	B	-
R32G32B32_SNORM	FLOAT	R	G	B	-
R32G32B32_SSCALED	FLOAT	R	G	B	-
R32G32B32_USCALED	FLOAT	R	G	B	-
R32G32B32_SFIXED	FLOAT	R	G	B	-
R16G16B16A16_UNORM	FLOAT	R	G	B	A
R16G16B16A16_SNORM	FLOAT	R	G	B	A
R16G16B16A16_SINT	SINT	R	G	B	A
R16G16B16A16_UINT	UINT	R	G	B	A
R16G16B16A16_FLOAT	FLOAT	R	G	B	A
R32G32_FLOAT	FLOAT	R	G	-	-
R32G32_SINT	SINT	R	G	-	-

Source Element	Converted Component				
R32G32_UINT	UINT	R	G	-	-
R32G32_UNORM	FLOAT	R	G	-	-
R32G32_SNORM	FLOAT	R	G	-	-
R64_FLOAT	FLOAT	R	-	-	-
R16G16B16A16_SSCALED	FLOAT	R	G	B	A
R16G16B16A16_USCALED	FLOAT	R	G	B	A
R32G32_SSCALED	FLOAT	R	G	-	-
R32G32_USCALED	FLOAT	R	G	-	-
R32G32_SFIXED	FLOAT	R	G	-	-
R64_PASSTHRU	NONE	R	-	-	-
B8G8R8A8_UNORM	FLOAT	B	G	R	A
R10G10B10A2_UNORM	FLOAT	R	G	B	A
R10G10B10A2_UINT	UINT	R	G	B	A
R10G10B10_SNORM_A2_UNORM	FLOAT	R	G	B	A
R8G8B8A8_UNORM	FLOAT	R	G	B	A
R8G8B8A8_SNORM	FLOAT	R	G	B	A
R8G8B8A8_SINT	SINT	R	G	B	A
R8G8B8A8_UINT	UINT	R	G	B	A
R16G16_UNORM	FLOAT	R	G	-	-
R16G16_SNORM	FLOAT	R	G	-	-
R16G16_SINT	SINT	R	G	-	-
R16G16_UINT	UINT	R	G	-	-
R16G16_FLOAT	FLOAT	R	G	-	-
B10G10R10A2_UNORM	FLOAT	R	G	B	A
R11G11B10_FLOAT	FLOAT	R	G	B	-
R32_SINT	SINT	R	-	-	-
R32_UINT	UINT	R	-	-	-
R32_FLOAT	FLOAT	R	-	-	-
R32_UNORM	FLOAT	R	-	-	-
R32_SNORM	FLOAT	R	-	-	-
R10G10B10X2_USCALED	FLOAT	R	G	B	-
R8G8B8A8_SSCALED	FLOAT	R	G	B	A
R8G8B8A8_USCALED	FLOAT	R	G	B	A
R16G16_SSCALED	FLOAT	R	G	-	-
R16G16_USCALED	FLOAT	R	G	-	-
R32_SSCALED	FLOAT	R	-	-	-
R32_USCALED	FLOAT	R	-	-	-

Source Element	Converted Component				
R8G8_UNORM	FLOAT	R	G	-	-
R8G8_SNORM	FLOAT	R	G	-	-
R8G8_SINT	SINT	R	G	-	-
R8G8_UINT	UINT	R	G	-	-
R16_UNORM	FLOAT	R	-	-	-
R16_SNORM	FLOAT	R	-	-	-
R16_SINT	SINT	R	-	-	-
R16_UINT	UINT	R	-	-	-
R16_FLOAT	FLOAT	R	-	-	-
R8G8_SSCALED	FLOAT	R	G	-	-
R8G8_USCALED	FLOAT	R	G	-	-
R16_SSCALED	FLOAT	R	-	-	-
R16_USCALED	FLOAT	R	-	-	-
R8_UNORM	FLOAT	R	-	-	-
R8_SNORM	FLOAT	R	-	-	-
R8_SINT	SINT	R	-	-	-
R8_UINT	UINT	R	-	-	-
R8_SSCALED	FLOAT	R	-	-	-
R8_USCALED	FLOAT	R	-	-	-
R8G8B8_UNORM	FLOAT	R	G	B	-
R8G8B8_SNORM	FLOAT	R	G	B	-
R8G8B8_SSCALED	FLOAT	R	G	B	-
R8G8B8_USCALED	FLOAT	R	G	B	-
R8G8B8_SINT	SINT	R	G	B	-
R8G8B8_UINT	UINT	R	G	B	-
R8G8B8_UINT	UINT	R	G	B	-
R64G64B64A64_FLOAT	FLOAT	R	G	B	A
R64G64B64_FLOAT	FLOAT	R	G	B	A
R16G16B16_FLOAT	FLOAT	R	G	B	-
R16G16B16_UNORM	FLOAT	R	G	B	-
R16G16B16_SNORM	FLOAT	R	G	B	-
R16G16B16_SSCALED	FLOAT	R	G	B	-
R16G16B16_USCALED	FLOAT	R	G	B	-
R16G16B16_UINT	UINT	R	G	B	-
R16G16B16_SINT	SINT	R	G	B	-
R32_SFIXED	FLOAT	R	-	-	-
R10G10B10A2_SNORM	FLOAT	R	G	B	A

Source Element	Converted Component				
R10G10B10A2_USCALED	FLOAT	R	G	B	A
R10G10B10A2_SSCALED	FLOAT	R	G	B	A
R10G10B10A2_SINT	SINT	R	G	B	A
B10G10R10A2_SNORM	FLOAT	R	G	B	A
B10G10R10A2_USCALED	FLOAT	R	G	B	A
B10G10R10A2_SSCALED	FLOAT	R	G	B	A
B10G10R10A2_UINT	UINT	R	G	B	A
B10G10R10A2_SINT	SINT	R	G	B	A
R64G64B64A64_PASSTHRU	NONE	R	G	B	A
R64G64B64_PASSTHRU	NONE	R	G	B	-

DestinationFormatSelection

The **Component Select 0..3** bits are then used to select, on a per-component basis, which destination components will be written and with which value. The supported selections are the converted source component, VertexID, InstanceID, PrimitiveID, the constants 0 or 1.0f, or nothing (VFCOMP_NO_STORE). If a converted component is listed as '-' (not available) in the "Source Element Formats" table (above). It must not be selected (via VFCOMP_STORE_SRC), or an UNPREDICTABLE value will be stored in the destination component.

The selection process sequences from component 0 to 3. Once a **Component Select** of VFCOMP_NO_STORE is encountered, all higher-numbered **Component Select** settings must also be programmed as VFCOMP_NO_STORE. It is therefore not permitted to have 'holes' in the destination VE.

Dangling Vertex Removal

The last functional stage of processing of the 3DPRIMITIVE command is the removal of "dangling" vertices. This stage includes the discarding of primitive topologies without enough vertices for a single object (e.g., a TRISTRIP with only two vertices), as well as the discarding of trailing vertices that do not form a complete primitive (e.g., the last two vertices of a 5-vertex TRILIST). 3D APIs typically require these vertices to be (effectively) discarded before the VS stage.

Statistics Gathering

This function is best described as a filter operating on the vertex stream emitted from the processing of the 3DPRIMITIVE. The filter inputs the PrimType, PrimStart, and PrimEnd values associated with the generated vertices. The filter only outputs primitive topologies without dangling vertices. This requires the filter to (a) be able to buffer some number of vertices, and (b) be able to remove dangling vertices from the pipeline and dereference the associated VUE handles.

3DSTATE_VF_STATISTICS

Vertices Generated

VF will increment the IA_VERTICES_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) for each vertex it fetches, even if that vertex comes from a cache rather than directly from a vertex buffer in memory. Any "dangling" vertices (fetched vertices that are part of an incomplete object) will not be included.

Objects Generated

VF will increment the IA_PRIMITIVES_COUNT Register (see Memory Interface Registers in vol1a System Overview) for each object (point, line, triangle, or quadrilateral) that it forwards down the pipeline.

For LINELOOP, the last (closing) line object is counted.

Vertex Shader (VS) Stage

The Vertex Shader (VS) stage of the 3D Pipeline is used to perform processing ("shading") of vertices after they are assembled and written to the URB by the VF function. The primary function of the VS stage is to pass vertices that miss in the VS Cache to VS threads, and then pass the VS thread-generated vertices down the pipeline. Vertices that hit in the VS Cache have already been shaded and are therefore passed down the pipeline unmodified.

When the VS stage is disabled, vertices flow through the unit unmodified (i.e., as written by the VF unit).

State

Register
3DSTATE_VS
3DSTATE_CONSTANT_VS
3DSTATE_PUSH_CONSTANT_ALLOC_VS
3DSTATE_BINDING_TABLE_POINTERS_VS
3DSTATE_SAMPLER_STATE_POINTERS_VS
3DSTATE_URB_VS

Functions

Vertex Shader Cache (VS$)

Note: The VS$ should not be confused with input data caches used by the VF stage when fetching data from index or vertex buffers in memory.

The 3D Pipeline employs a Vertex Shader Cache (VS$) that is shared between the VF and VS stages. (See *Vertex Fetch* chapter for additional information). The vertex index generated by the VF stage is used as the cache tag. The cached data contains the URB handle of a VUE, which in turn typically contains the vertex data output from a previously executed VS shader, though if the VS function is disabled the VUE will contain the input vertex data generated by the VF stage.

When the VF stage processes a vertex, it will first perform a lookup in the VS$. If the vertex hits in the VS$, the VS stage will return the hit VUE handle to the VF stage, and the VF stage will subsequently pass the returned VUE handle back down the FF pipeline to VS. If the vertex misses in the VS$ (or always, if the VS$ is disabled), the VS stage will allocate a VUE handle for the miss vertex and return this to the VF stage. The VF stage will then proceed to fetch/generate the input vertex data, store the results into the VUE, and then pass the VUE down to the VS stage. If the VS function is enabled, the VUE handle/data will be used as input to a VS shader thread, and that thread will overwrite the VUE with the shader results.

The VS$ may be explicitly DISABLED via the Vertex Cache Disable bit in 3DSTATE_VS. Even when explicitly ENABLED, the VS stage will (by default) implicitly disable the VS$ whenever it detects one of the following conditions:

Sequential indices are used in the 3DPRIMITIVE command (though this is effectively a don't care as there would not be any VS$ hits).

The implicit disable persists as long as one of these conditions persist, after which the VS$ is invalidated.

The VS$ is implicitly invalidated between 3DPRIMITIVE commands and between instances within a 3DPRIMITIVE command - therefore use of InstanceID in a Vertex Element is not a condition under which the cache is implicitly disabled.

The following table summarizes the modes of operation of the VS$.

VS$	VS Function Enable	Mode of Operation
DISABLED (implicitly or explicitly)	DISABLED	The VS$ is not used. VF stage assembles all vertices and writes them into the VUE supplied by the VS stage. VS stage subsequently passes references to these VUEs down the pipeline without spawning any VS threads.
	ENABLED	The VS$ is not used. VF stage assembles all vertices and writes them into the VUE supplied by the VS stage. VS stage subsequently spawns VS threads to process all vertices, overwriting the input data with the results. The VS stage pass references to these VUEs down the pipeline. Usage Model: This mode is only used when the VS function is required, but either (a) the VS kernel produces a side effect (e.g., writes to a memory buffer) which in turn requires every vertex to be processed by a VS thread.
ENABLED	DISABLED	The VS$ is used to provide reuse of VF-generated vertices. The VF stage checks the cache and only processes (assembles/writes) vertices that miss in the VS$. In either case, the VS stage passes references to vertices (that hit or miss) down the pipeline without spawning any VS threads. Usage Model: Normal operation when the VS function is not required (e.g., SW has detected a VS shader that simply copies outputs to inputs).

VS$	VS Function Enable	Mode of Operation
	ENABLED	The VS$ is used to provide reuse of VS-processed vertices. The VF stage checks the cache and only processes (assembles/writes) vertices that miss in the VS$. The VS stage only processes (shades) the vertices that missed in the VS$. The VS stage sends references to hit or missed vertices down the pipeline in the correct order. Usage Model: Normal operation when the VS function is required and use of the VS$ is permissible.

VS Thread Dispatch Masks

The VS stage controls the initial value loaded into the EU's Dispatch Mask state register as part of thread dispatch.

SIMD8 Dispatch Mask

In SIMD8 dispatch mode, the EU Dispatch Mask is initialized as a function of the number of vertices included in the thread dispatch, as follows:

- 1 vertex: 0x00000001
- 2 vertices: 0x00000003
- 3 vertices: 0x00000007
- 4 vertices: 0x0000000F
- 5 vertices: 0x0000001F
- 6 vertices: 0x0000003F
- 7 vertices: 0x0000007F
- 8 vertices: 0x000000FF

Vertex Output

VS threads must always write the destination URB entries whose handles are passed in the thread payload. Refer to Vertex Data Overview for details on any required contents/formats.

Thread Termination

VS threads must signal thread termination, in all likelihood on the last message output to the URB shared function. Refer to the *ISA* doc for details on End-Of-Thread indication.

Primitive Output

The VS unit will produce an output vertex reference for every input vertex reference received from the VF unit, in the order received. The VS unit simply copies the PrimitiveType, StartPrim, and EndPrim information associated with input vertices to the output vertices and does not use this information in any way. Neither does the VS unit perform any readback of URB data.

Statistics Gathering

The VS stage tracks a single pipeline statistic, the number of times a vertex shader is executed. A vertex shader is executed for each vertex that is fetched on behalf of a 3DPRIMITIVE command, unless the shaded results for that vertex are already available in the vertex cache. If the **Statistics Enable** bit in VS_STATE is set, the VS_INVOCATION_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) will be incremented for *each vertex* that is dispatched to a VS thread.

When **VS Function Enable** is DISABLED and **Statistics Enable** is ENABLED, VS_INVOCATION_COUNT increments by one for every vertex that passes through the VS stage, even though no VS threads are spawned.

Payloads

SIMD8 Payload

The following table describes the payload delivered to VS threads.

SIMD8 VS Thread Payload

DWord	Bits	Description
R0.7	31	
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Description** **Scratch Space Buffer.** Specifies the index of the scratch buffer allocated to the stage. See **Scratch Space Buffer** description in VS_STATE. Format = SurfaceStateOffset[27:6]
R0.5	9:0	**Description** **FFTID:** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Reserved for HW Implementation Use. **Format:** U10 **Range:** 0-727

DWord	Bits	Description
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **General State Base Address** or **Dynamic State Base Address**. Format = DynamicStateOffset[31:5]
	4	**Description** **Single Instance.** If set, all valid vertices included in the thread payload come from the same instance of a 3DPRIMITIVE command. Otherwise, the vertices come from more than one instance. When **SIMD8SingleInstanceDispatchEnable** is ENABLED, this bit will (by definition) always be set.
	3:0	**Description** Reserved.
R0.2 : R0.0	31:0	Reserved: MBZ
R1.7	31:0	Vertex 7 URB Return Handle (see R1.0)
R1.6	31:0	Vertex 6 URB Return Handle (see R1.0)
R1.5	31:0	Vertex 5 URB Return Handle (see R1.0)
R1.4	31:0	Vertex 4 URB Return Handle (see R1.0)
R1.3	31:0	Vertex 3 URB Return Handle (see R1.0)
R1.2	31:0	Vertex 2 URB Return Handle (see R1.0)
R1.1	31:0	Vertex 1 URB Return Handle (see R1.0)
R1.0	31:16	Reserved
	15:0	**Vertex 0 URB Return Handle.** This is the offset within the URB where Vertex 0 is to be stored. Format: 64B-granular offset into the URB This is an offset into the Local URB.
[Varies] optional	255:0	**Constant Data (optional):** Please refer to the Push Constants chapter in the General Programming of Thread-Generating Stages section for more details on size and source of constant data.

DWord	Bits	Description
		Vertex Data: Input data for the 8 input vertices is located here. Vertex0 data is passed in DW0 of these GRFs, and Vertex 7 data is passed in DW7. The first GRF contains Element 0 Component 0 for all 8 vertices, followed by components 1-3 in the three subsequent GRFs. This is followed by GRFs containing Element 1, and so on, up to the number of elements specified by Vertex URB Read Length. Note that the maximum limit is 30 elements per vertex, though the practical limit imposed by the compiler is likely lower.
Rv.7	31:0	Vertex 7 Element 0 Component 0
Rv.6	31:0	Vertex 6 Element 0 Component 0
Rv.5	31:0	Vertex 5 Element 0 Component 0
Rv.4	31:0	Vertex 4 Element 0 Component 0
Rv.3	31:0	Vertex 3 Element 0 Component 0
Rv.2	31:0	Vertex 2 Element 0 Component 0
Rv.1	31:0	Vertex 1 Element 0 Component 0
Rv.0	31:0	Vertex 0 Element 0 Component 0
Rv+1.7	31:0	Vertex 7 Element 0 Component 1
Rv+1.6	31:0	Vertex 6 Element 0 Component 1
Rv+1.5	31:0	Vertex 5 Element 0 Component 1
Rv+1.4	31:0	Vertex 4 Element 0 Component 1
Rv+1.3	31:0	Vertex 3 Element 0 Component 1
Rv+1.2	31:0	Vertex 2 Element 0 Component 1
Rv+1.1	31:0	Vertex 1 Element 0 Component 1
Rv+1.0	31:0	Vertex 0 Element 0 Component 1
..		Vertex 0-7 Element 0 Component 2,3
..		Vertex 0-7 Element 1 Component 0-3
..		Vertex 0-7 Element 2-N Component 0-3

Hull Shader (HS) Stage

The Hull Shader (HS) stage of the pipeline is used to process patchlist (PATCHLIST_*n*) topologies in support of higher-order surface (HOS) tessellation. If the HS stage is enabled, each incoming patch object is processed by a possible series of HS threads. The combined output of these threads is a Patch URB Entry ("patch record") written to the URB. This patch record is used by subsequent stages (TE, DS) to complete the HOS tessellation operations.

Programming Note
When MeshShading is Enabled, only TASK-related state is used by HW and all HS-related state is IGNORED by HW.

The vertices associated with patchlist primitives are also referred to as "Input Control Points" (ICPs) to contrast them with any "Output Control Points" the HS threads may write to the patch record. (The definition and use of OCPs are outside the scope of this document).

The HS stage also performs statistics counting. Incomplete topologies do not reach the HS stage.

The HS, TE, and DS stages must be enabled and disabled together. When these stages are disabled, all topologies (including patchlist topologies) simply pass through to the GS stage. When these stages are enabled, only patchlist topologies should be issued to the pipeline, otherwise behavior is UNDEFINED.

State

This section contains the state registers for the Hull Shader

Register
3DSTATE_HS
3DSTATE_PUSH_CONSTANT_ALLOC_HS
3DSTATE_CONSTANT_HS
3DSTATE_CONSTANT(Body)
3DSTATE_BINDING_TABLE_POINTERS_HS
3DSTATE_SAMPLER_STATE_POINTERS_HS
3DSTATE_URB_HS

Programming Note
When MeshShading is Enabled, only TASK-related state is used by HW and all HS-related state is IGNORED by HW.

Functions

Patch Object Staging

The HS unit accepts patchlist topologies as a stream of incoming vertices. Depending on the number of vertices per patch object (as specified by the PATCHLIST_*n* topology), the HS thread assembles each complete patch object and passes it (its vertices, PrimitiveID, etc.) to HS thread(s) as described below.

HS Thread Execution

Input to HS threads is comprised of:

- Input Control Points (incoming patch vertices), pushed into the payload and/or passed indirectly via URB handles.
- Push Constants (common to all threads)
- Patch Data handle
- Resources available via binding table entries (accessed through shared functions)
- Miscellaneous payload fields (Instance Number, etc.)

Typically, the only output of the HS threads is the Patch URB Entry (patch record). All thread instances for an input patch are passed the same patch record handle. As the (possibly concurrent) threads can both read and write the patch record, it is up to the kernels to ensure deterministic results. One approach would be to use the thread's Instance Number as an index for URB write destinations.

HS Thread Dispatch Mask

The HS stage controls the initial value loaded into the EU's Dispatch Mask state register as part of thread dispatch.

SINGLE_PATCH Dispatch Mask

In SINGLE_PATCH mode, the EU Dispatch Mask is initialized at thread dispatch to 0x000000FF.

8_PATCH Dispatch Mask

In 8_PATCH mode, the EU Dispatch Mask is initialized as a function of the number of patches included in the thread dispatch, as follows:

- 1 patch: 0x00000001
- 2 patches 0x00000003
- 3 patches: 0x00000007
- 4 patches: 0x0000000F
- 5 patches: 0x0000001F
- 6 patches: 0x0000003F
- 7 patches: 0x0000007F
- 8 patches: 0x000000FF

Patch URB Entry (Patch Record) Output

For each patch, the HS thread(s) generate a single patch record, starting with a fixed 32B Patch Header.

When the final thread instance terminates, the patch record handle is passed down the pipeline to the Tessellation Engine (TE).

Patch Header DW0-7

The first eight DWords of the Patch URB Entry are defined as a "Patch Header". A Patch Header is written by an HS thread and read by the TE stage. It normally contains up to six **Tessellation Factors** (TFs) that determine how finely the TE stage needs to tessellate a domain (if at all).

The layout of the patch header depends on the **TE Domain** and **PatchHeaderLayout.**

The **PatchHeaderLayout** of REVERSED_TRI_INSIDE_SEPARATE can only be used with a **TE Domain** of TRI.

Patch Header: QUAD Domain / LEGACY Patch Header Layout

DWord	Bits	Description
7	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32
6	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
5	31:0	**UEQ1 Tessellation Factor** Format: FLOAT32
4	31:0	**VEQ1 Tessellation Factor** Format: FLOAT32
3	31:0	**Inside U Tessellation Factor** Format: FLOAT32
2	31:0	**Inside V Tessellation Factor** Format: FLOAT32
1-0	31:0	Reserved : MBZ

Patch Header: QUAD Domain / REVERSED Patch Header Layout

DWord	Bits	Description
7:6	31:0	Reserved : MBZ
5	31:0	**Inside V Tessellation Factor** Format: FLOAT32
4	31:0	**Inside U Tessellation Factor** Format: FLOAT32
3	31:0	**VEQ1 Tessellation Factor** Format: FLOAT32
2	31:0	**UEQ1 Tessellation Factor** Format: FLOAT32
1	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
0	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32

Patch Header: TRI Domain / LEGACY Patch Header Layout

DWord	Bits	Description
7	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32
6	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
5	31:0	**WEQ0 Tessellation Factor** Format: FLOAT32
4	31:0	**Inside Tessellation Factor** Format: FLOAT32
3-0	31:0	Reserved : MBZ

Patch Header: TRI Domain / REVERSED Patch Header Layout

DWord	Bits	Description
7-4	31:0	Reserved : MBZ
3	31:0	**Inside Tessellation Factor** Format: FLOAT32
2	31:0	**WEQ0 Tessellation Factor** Format: FLOAT32
1	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
0	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32

Patch Header: TRI Domain / REVERSED_TRI_INSIDE_SEPARATE Patch Header Layout

DWord	Bits	Description
7-5	31:0	Reserved : MBZ
4	31:0	**Inside Tessellation Factor** Format: FLOAT32
3	31:0	Reserved : MBZ
2	31:0	**WEQ0 Tessellation Factor** Format: FLOAT32
1	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
0	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32

Patch Header: ISOLINE Domain / LEGACY Patch Header Layout

DWord	Bits	Description
7	31:0	**Line Detail Tessellation Factor** Format: FLOAT32
6	31:0	**Line Density Tessellation Factor** Format: FLOAT32
5-0	31:0	Reserved : MBZ

Patch Header: ISOLINE Domain / REVERSED Patch Header Layout

DWord	Bits	Description
7-2	31:0	Reserved : MBZ
1	31:0	**Line Density Tessellation Factor** Format: FLOAT32
0	31:0	**Line Detail Tessellation Factor** Format: FLOAT32

Statistics Gathering

HS Invocations

The HS unit controls the HS_INVOCATIONS counter, which counts the number of patches processed by the HS stage.

Payloads

8_PATCH Payload

The following table shows the layout of the payload delivered to HS threads. Refer to 3D Pipeline Stage Overview (*3D Pipeline*) for details on those fields that are common amongst the various pipeline stages.

Patch object vertex (ICP) data can be passed by value (data pushed in the payload) and/or by reference (URB handle pushed in the payload).

8_PATCH HS Thread Payload

GRF DWord	Bits	Description
R0.7	31	
	30:0	Reserved
R0.6	31	**Early Dereference Enable**: The enabling of early dereference. Reserved for Implementation Use
	30:24	Reserved

GRF DWord	Bits	Description
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the scratch space allocated to this thread, specified as a 1KB-aligned offset from the General State Base Address. Format = GeneralStateOffset[31:10]
	9	Reserved
	8:0	**FFTID.** This ID is assigned by the fixed function unit and is a relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format = SurfaceStateOffset[31:5]
	4:0	**Reserved**
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the Dynamic State Base Address. Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	Reserved
R0.2	31:24	**Instance Count:** This field specifies the number of threads spawned per input patch. Format = U5 Range = [1,32] Reserved for Implementation Use
	14:11	**Reserved**
	9	**Reserved**
	8	Barrier Enable
	7:0	**Instance Number.** A patch-relative instance number between 0 and InstanceCount-1 Format = U7
R0.1	31:0	Reserved
R0.0	31:16	Reserved

GRF DWord	Bits	Description
	15:6	**Early Dereference ID**: The ID of the thread used for early dereference. Reserved for Implementation Use
	5:0	Reserved
R1.7	31:0	**URB Return Handle for Patch 7** (See R1.0)
R1.6	31:0	**URB Return Handle for Patch 6** (See R1.0)
R1.5	31:0	**URB Return Handle for Patch 5** (See R1.0)
R1.4	31:0	**URB Return Handle for Patch 4** (See R1.0)
R1.3	31:0	**URB Return Handle for Patch 3** (See R1.0)
R1.2	31:0	**URB Return Handle for Patch 2** (See R1.0)
R1.1	31:0	**URB Return Handle for Patch 1** (See R1.0)
R1.0	31:16	Reserved
	15:0	**URB Return Handle 0:** This is the offset of the Patch 0's URB entry, where shading results are to be written. Format: U16 64B-aligned URB Offset This is a Local URB Offset.
The following register is included only if Include PrimitiveID is enabled.		
R2.7	31:0	**Primitive ID 7.** This field contains the Primitive ID associated with Patch 7 Format: U32
R2.6	31:0	**Primitive ID 6.** This field contains the Primitive ID associated with Patch 6 Format: U32
R2.5	31:0	**Primitive ID 5.** This field contains the Primitive ID associated with Patch 5 Format: U32
R2.4	31:0	**Primitive ID 4.** This field contains the Primitive ID associated with Patch 4 Format: U32
R2.3	31:0	**Primitive ID 3.** This field contains the Primitive ID associated with Patch 3 Format: U32
R2.2	31:0	**Primitive ID 2.** This field contains the Primitive ID associated with Patch 2 Format: U32
R2.1	31:0	**Primitive ID 1.** This field contains the Primitive ID associated with Patch 1

GRF DWord	Bits	Description
		Format: U32
R2.0	31:0	**Primitive ID 0.** This field contains the Primitive ID associated with Patch 0 Format: U32
The following registers are included only if Include Vertex Handles is enabled		
Rn.7	31:16	Reserved
	15:0	**Patch 7 ICP 0 Handle**
Rn.6	31:16	Reserved
	15:0	**Patch 6 ICP 0 Handle**
Rn.5	31:16	Reserved
	15:0	**Patch 5 ICP 0 Handle**
Rn.4	31:16	Reserved
	15:0	**Patch 4 ICP 0 Handle**
Rn.3	31:16	Reserved
	15:0	**Patch 3 ICP 0 Handle**
Rn.2	31:16	Reserved
	15:0	**Patch 2 ICP 0 Handle**
Rn.1	31:16	Reserved
	15:0	**Patch 1 ICP 0 Handle**
Rn.0	31:16	Reserved
	15:0	**Patch 0 ICP 0 Handle**
[Rn+1]	255:0	**ICP 1 Handle for Patches 0-7**
[Rn+2]	255:0	**ICP 2 Handle for Patches 0-7**
...		...
[Rn+31]	255:0	ICP 31 Handle for Patches 0-7
[Varies] optional	255:0	**Constant Data (optional):** Please refer to the Push Constants chapter in the General Programming of Thread-Generating Stages section for more details on size and source of constant data.
Varies		**Pushed Vertex Data (optional)** Input data for the 8 patches is located here. Patch 0 (starting with Vertex 0 of Patch 0) data is passed in DW0 of these GRFs, and Patch 7 data is passed in DW7. The first GRF contains Vertex 0 Element 0 Component 0 for all 8 patches, followed by components 1-3 in the three subsequent GRFs. This is followed by GRFs containing Vertex 0 Element 1 (if it exists), and so on, up to the number of Vertex 0 elements specified by Vertex URB Read Length. This is followed by the data for Vertex 1 for all patches (if it exists), and so on until all relevant vertices are passed.

GRF DWord	Bits	Description
		Programming Note **Context:** 8_PATCH Payload - Pushed Vertex Data The amount of data passed is limited by the number of GRFs supported by EUs. Software is responsible for comprehending this limit and resorting to the pull model as required.
Rv.7	31:0	**Patch 7 Vertex 0 Element 0 Component 0**
Rv.6	31:0	**Patch 6 Vertex 0 Element 0 Component 0**
Rv.5	31:0	**Patch 5 Vertex 0 Element 0 Component 0**
Rv.4	31:0	**Patch 4 Vertex 0 Element 0 Component 0**
Rv.3	31:0	**Patch 3 Vertex 0 Element 0 Component 0**
Rv.2	31:0	**Patch 2 Vertex 0 Element 0 Component 0**
Rv.1	31:0	**Patch 1 Vertex 0 Element 0 Component 0**
Rv.0	31:0	**Patch 0 Vertex 0 Element 0 Component 0**
Rv+1	31:0	**Patch 0-7 Vertex 0 Element 0 Component 1**
...		and so on...

Task Shader (TASK) Stage

The primary function of the Task Shader stage is to request EU thread group dispatches for a series of TaskShader ThreadGroups (TaskTGs). Requests for TaskTG invocations originate directly from a 3DMESH command when TaskShaderEnabled.

Each TaskTG will be provided with an output TaskShader URB Entry (TUE) handle in the payload of each EU thread of the TaskTG dispatch. The threads of the TaskTG shall collaborate/coordinate the generation of Task data and its storage in the TUE. The TUE storage starts with a MeshTGLaunchCount by which the TaskTG specifies some number (possibly zero) of MeshTGs to be subsequently spawned in the downstream MeshShader stage. If the count is non-zero, the TUE handle will be passed in the payloads of those MeshTGs, allowing the rest of the TUE to pass arbitrary data structures from a TaskTG to the MeshTGs it requests to be spawned. These MeshTG requests will be passed down the pipeline in order of increasing TaskTG ID.

TaskTG requests for spawned MeshTGs may optionally be redistributed across all enabled GSlices prior to MeshShading (for performance purposes only, no functional impact). If Task Redistribution is disabled, the spawned MeshTGs will execute entirely on the same Gslice the TaskTG execution, which may lead to suboptimal load balancing and performance. Refer to programming details below.

Task Shader State

This section contains links to the state commands for the Task Shader stage.

Register
3DSTATE_TASK_CONTROL
3DSTATE_TASK_REDISTRIB
3DSTATE_TASK_SHADER
3DSTATE_TASK_SHADER_DATA
3DSTATE_URB_ALLOC_TASK

Task Shader Functions

This section describes the various functions performed by the Task Shader stage fixed function:

- In response to receipt of 3DMESH operations with TaskShaderEnabled:
  - Task URB Entries are allocated in order to pass TUE output handles in TaskShader thread payloads
  - TaskShader thread requests are generated in order to initiate TaskTG dispatch and execution
  - TaskShader-specific statistics counters are incremented
  - If enabled, Task Redistribution is performed after TaskTGs EOT and are correctly ordered

Task Shader Thread Execution

Task Shader threads primarily execute as subsetted GPGPU Compute threads. Dispatch of Task Shader threads comes as a result of 3DMESH command execution, which requests the dispatch of a sequence of TaskShader ThreadGroups (TaskTGs).

Task Shader ThreadGroup Dispatch Dimension

ThreadGroup Dispatch Dimensions Supported
Only 1D TG dispatch is support by the 3DMESH command.
The 3DMESH command supports both 1D and 3D (X,Y,Z) TG dispatch requests, as specified by 3DMESH::MeshDimensionSelect (1D|3D). A 3D TG dispatch request will, however, be converted to a 1D TG dispatch, with the count of TGs equal to the product of the X, Y and Z ThreadGroup Counts. The requested dispatch X,Y,Z dimensions from the 3DMESH command are passed into the TaskShader thread payload, and can be used by the kernel to generate 3D TGIDs from the 1D ThreadGroup ID also passed in the payload. This computation performed by the kernel can apply an arbitrary dispatch walk order (e.g., X-then-Y-then-Z, Z-then-X-then-Y, etc.). The following computation may be used for X-then-Y-then-Z walk order: TGID.Z = QUOTIENT(TGID_1D / DIM_X*DIM_Y) TEMP = REMAINDER(TGID_1D / DIM_X*DIM_Y) TGID.Y = QUOTIENT(TEMP / DIM_X) TGID.X = REMAINDER(TEMP / DIM_X)

The threads in a TaskTG can be provided with a BarrierID and SLM allocation, allowing GPGPU-like thread collaboration at a thread group level. Payload inputs to Task Shader threads are a subset of GPGPU Compute thread payload inputs and include per-lane 1-D Local ID values used to identify an API thread (lane) within the local threadgroup. Task Shader payloads may also include an Indirect Parameter Pointer as well as one GRF of Inline Parameter Data.

The subsetting of GPGPU Compute capabilities include (but not limited to):

- TaskShader thread dispatches are 1D only.
- TaskShaders only utilize Bindless surfaces and samplers. Binding Table and Sampler State Pointers are not available/supplied.
- TaskShader thread dispatches do not support the following GPGPU Compute features:
  - Tiled walks
  - Partitioning
  - Mid-Thread Preemption

Unlike GPGPU Compute threads, Task Shader threads are provided with an output Task URB Entry (TUE) Offset which points to a per-TaskTG URB allocation. The threads within a TaskTG are able to collaborate in order to write the required data into the TUE. The only required output is a MeshShaderLaunchCount DWord, with the remainder of the TUE available for optional SW-specific data structures provided as input to the requested MeshShader ThreadGroup dispatches (if any). The ability to pass this sort of temporary data via internal storage between different types of shaders is unique to Task-->Mesh Shaders and not available to GPGPU Compute threads (which can only pass outputs via memory). Task Shaders are also able to output via memory, though like GPGPU Compute threads there is no HW-managed data passing (and allocation/deallocation bookkeeping) to subsequent threads.

Task Shader URB Entry (TUE)

The Task Shader stage allocates an output TaskShader URB Entry (TUE) handle for each TaskShader ThreadGroup dispatched, and the output TUE handle is passed in each thread's payload. The table below describes the contents of the TUE.

Task Shader URB Entry

<table>
<tr><th>DWord</th><th>Contents</th></tr>
<tr><td>0</td><td>MeshShaderTGLaunchCount: This value allows a TaskShader TG to request the subsequent dispatch of some number of MeshShader TGs. If the count is non-zero, each of the dispatched MeshShader TGs will be passed the TUE handle associated with the TaskShader TG. If the count is zero, no MeshShader TGs will be dispatched.
<table><tr><th>Programming Note</th></tr><tr><td>This value shall equal the product of the Dispatch Dimension X, Dispatch Dimension Y and Dispatch Dimension Y values (DWord1,2,3).</td></tr></table></td></tr>
<tr><td>1</td><td>Dispatch Dimension X:</td></tr>
</table>

DWord	Contents
	This value specifies the number of MeshShaderTGs dispatched "in the X dimension". The value will be passed as DISP_DIM_X in the R0 Header of the MeshShaderTG threads and can be used by the kernel (in combination with the other Dispatch Dimension values) to generate a 3D ThreadGroup ID (TGID) from the 1D ThreadGroup ID value.
2	**Dispatch Dimension Y:** This value specifies the number of MeshShaderTGs dispatched "in the Y dimension". The value will be passed as DISP_DIM_Y in the R0 Header of the MeshShaderTG threads and can be used by the kernel (in combination with the other Dispatch Dimension values) to generate a 3D ThreadGroup ID (TGID) from the 1D ThreadGroup ID value If 3DMESH_1D is used to create the MESH, this Dword must be a value of 1.
3	**Dispatch Dimension Z:** This value specifies the number of MeshShaderTGs dispatched "in the Z dimension". The value will be passed as DISP_DIM_Z in the R0 Header of the MeshShaderTG threads and can be used by the kernel (in combination with the other Dispatch Dimension values) to generate a 3D ThreadGroup ID (TGID) from the 1D ThreadGroup ID value If 3DMESH_1D is used to create the MESH, this Dword must be a value of 1.
... up to URB Entry Size-1	**Remainder of URB Entry**: The remainder of the TUE may contain an arbitrary amount of SW-defined data passed between a TaskShader TGs and the subsequent MeshShader TGs it requests to be dispatched. HW provides the kernels access to this data, but otherwise ignores it. **Programming Note** It is suggested that SW reserve the 16 bytes following the TUE Header, and therefore start the SW-defined data structure at 32B alignment. This allows the TUE Header to always be written as 32 bytes with 32B alignment, the most optimal write performance case.

Task Shader Statistics

The **TASK_INVOCATION_COUNT** MMIO register in each GSlice accumulates API-level Task Shader invocations dispatched by the pipeline. For each TaskShader ThreadGroup dispatched, this register is incremented by the thread group size. This register does not count EU thread dispatches.

SW shall comprehend that a pipeline flush is required to ensure that preceding TaskShader work is included in the register value.

Task Redistribution

By default (i.e., 3DSTATE_TASK_REDISTRIB::TaskRedistributionMode == OFF), all the MeshTG dispatch requests generated by a batch of TaskTGs (BOT) executed by the Task stage of a given geometry pipeline will be restricted to execute on the lower portion of that same geometry pipeline. As geometry slice

compute (EU) resources and pre-raster-crossbar buffering capacity are limited, restricting a large number of MeshTGs to execute on only one geometry slice can lead to load imbalance and therefore risk a multi-slice configuration approaching single-slice performance. Therefore, the ability to redistribute large numbers of MeshTGs across all enabled geometry slices is desired.

(Note that the Task Redistribution controls described below are similar to the controls provided for geometry distribution across the upper geometry pipes, as specified in the 3DSTATE_VFG command.)

The 3DSTATE_TASK_REDISTRIB command is provided to enable and control the redistribution of MeshTG dispatch requests across the device. When 3DSTATE_TASK_REDISTRIB::**TaskRedistributionMode** == OFF, the default behavior is to keep MeshTG work local to the geometry slice, as described above. When 3DSTATE_TASK_REDISTRIB::**TaskRedistributionMode** != OFF, MeshTG dispatch requests may be redistributed across the (enabled) lower geometry pipes of the device as described below. Also, refer to the 3DSTATE_TASK_REDISTRIB command for more details.

Task Redistribution is enabled by setting 3DSTATE_TASK_REDISTRIB::TaskRedistributionMode to either RR_STRICT or RR_FREE. Both enable Task Redistribution across the enabled lower geometry pipes in the device in a Round-Robin (RR) fashion. **RR_FREE** mode skips over geometry slices that cannot accept additional work at the time of redistribution and offers improved load-balancing and typically the highest performance. **RR_STRICT** will maintain strict round-robin behavior, possibly waiting for the next sequential geometry pipeline to accept a redistribution request even while another geometry pipeline could accept that request. This mode may lead to non-optimal performance but can offer deterministic work distribution across the pipelines (when coupled with RR_STRICT distribution of work across the upper pipelines). This behavior can be helpful for performance analysis.

The granularity of Task Redistribution can be controlled by additional fields in 3DSTATE_TASK_REDISTRIB:

- 3DSTATE_TASK_REDISTRIB::**TaskRedistributionLevel** determines whether or not the MeshTG dispatch requests from a Task are first divided into batches of MeshTGs (BOMs) prior to possible redistribution. While **TASKREDISTRIB_BOM** enables this division, **TASKREDISTRIB_TASK** disables it and thus causes Task Redistribution to keep all MeshTG dispatch requests by a given Task combined (not first split into BOMs).
- 3DSTATE_TASK_REDISTRIB::**TargetMeshBatchSize** is used to specify a "target" value for TaskRedistribution granularity in units of MeshShader ThreadGroups (MeshTGs). This value is used to divide MeshTG dispatch requests into BOMs when TASKREDISTRIB_BOM is specified. This value is also used to define the BOM size unit used for the Local BOT Accumulator Threshold.

There are conditions underwhich MeshTG dispatch requests remain local to the issuing geometry slice. This behavior is provided to help avoid Task Redistribution overhead in runtime cases where redistribution is not necessarily required for optimal performance and may instead lower performance if redistribution is permitted.

- 3DSTATE_TASK_REDISTRIB::**LocalBOTAccumulatorThreshold** specifies an upper limit on the combined number of MeshTG dispatch requests across a batch of TaskTGs (BOT) that are to be executed on the local geometry slice. Only BOTs requesting a combined number of MeshTGs greater than the threshold value specified will be considered for Task Redistribution. This control therefore controls TaskRedistribution granularity at the BOT level and can be used to prevent redistribution altogether when a BOT requests only a few BOMs of MeshTGs.

- 3DSTATE_TASK_REDISTRIB::**SmallTaskThreshold** specifies a lower limit on the number of MeshTG dispatch requests that may be considered for TaskRedistribution. If a TaskTG requests MeshTG dispatches with a MeshTG count less than or equal to this value, those MeshTGs dispatch requests are strictly performed on the local geometry slice and not subjected to Task Redistribution. This control therefore controls TaskRedistribution at the Task level and can be used to prevent redistribution of a Task that requests a small number of MeshTG dispatches.

Task Shader Payloads

The TaskShader thread payload appears mostly as a subset of GPGPU Compute thread payloads, with the exception of the TaskShader URB Entry Offset. Additional information can therefore be found in the relevant GPGPU Compute thread payload subsections.

As illustrated below, the TaskShader thread payload is comprised of two or three sections.

- **R0 Header** (contents specified below)
- Optional, Per-lane, 16-bit **Local_ID.X** values, passed in R1 for SIMD8 or SIMD16 dispatch or R1,R2 for SIMD32 dispatch. The value for a given lane identifies the local thread (lane) position within the API threadgroup (which may include multiple EU thread dispatches).
- Optional, 1-GRF **Inline Parameter**, included if 3DSTATE_TASK_SHADER::EmitInlineParameter is set. This will be passed immediately following the Local_ID section (in R2 for SIMD8 or SIMD16, or in R3 for SIMD32). When enabled, SW shall provide the inline data via 3DSTATE_TASK_SHADER_DATA::InlineData state.

SIMD8,SIMD16 Dispatch

R0	R0 Header
R1	Local_ID.X[0-7 or 0-15]
R2	Inline Parameter (optional)

SIMD32 Dispatch

R0	R0 Header
R1	Local_ID.X[0-15]
R2	Local_ID.X[16-31]
R3	Inline Parameter (optional)

The **RO Header** of the Thread Dispatch Payload for a Task Shader thread:

DWord	Bits	Description
R0.7	31:0	Reserved: MBZ
R0.6	31:16	Reserved
	31:16	**DISP_MIN_X**: This field contains the Dispatch Dimension X value provided in the provoking 3DMESH_3D command. For 3DMESH_1D, the contents of the field are UNDEFINED. Format: U16
	15:0	**Task Shader URB Entry Offset:** This is the offset of the thread group's output Task Shader URB Entry (TUE) within the Slice's Local URB. Format: U16 64B-aligned Local URB Offset
R0.5	31:10	**Scratch Space Buffer.** Specifies the surface state index to the Scratch Buffer for use by the kernel. This surface state index is relative to the **Surface State Base Address**. Format = SurfaceStateOffset[27:6]
	9:0	**FFTID.** This ID is assigned by HW and is a unique identifier for the thread in comparison to other concurrent threads (of any thread group). It is used to free up resources used by the thread upon thread completion. Format = U9
R0.4	31:0	Reserved: MBZ
	31:16	**DISP_MIN_Z**: This field contains the Dispatch Dimension Z value provided in the provoking 3DMESH_3D command. For 3DMESH_1D, the contents of the field are UNDEFINED. Format: U16
	15:0	**DISP_MIN_Y**: This field contains the Dispatch Dimension Y value provided in the provoking 3DMESH_3D command. For 3DMESH_1D, the contents of the field are UNDEFINED. Format: U16
R0.3	31:0	Reserved: MBZ
	31:0	**Extended Parameter 0 (XP0)**: When 3DSTATE_MESH_SHADER::XP0Required is set, this field is a copy of the Extended Parameter 0 value from the provoking 3DMESH command (if 3DMESH::ExtendedParameter0Present was set) or zero (if 3DMESH::ExtendedParameter0Present was clear). If 3DSTATE_MESH_SHADER::XP0Required is clear, this field is UNDEFINED. Format U32
R0.2	31:24	**Number of Threads in GPGPU Thread Group.** This field specifies the number of EU threads that are in each TaskShader thread group.
	14:11	Reserved: MBZ

DWord	Bits	Description
	9	Reserved: MBZ
	8	**Barrier Enable.** This field indicates that a barrier has been allocated for this kernel.
	7:0	**Thread Number Within Thread Group**. This field is used to identify a particular thread within a thread group. Format: U8
R0.1	31:0	**Thread Group ID X:** This field identifies the X component of the thread group that this thread belongs to.
R0.0	31:5	**Indirect Data Start Address**: This field specifies the Graphics Memory starting address of the data to be loaded into the kernel for processing. This pointer is relative to the **General State Base Address.** It is the 64-byte aligned address of the indirect data. Format: GeneralStateOffset[31:6]
	4:1	Reserved: MBZ
	0	**Local ID Present:** Indicates that the HW-generated Local ID X follows the payload.

Tessellation Engine (TE) Stage

When enabled, the Tessellation Engine (TE) stage performs tessellation of incoming patches (decomposition of patches into a set of smaller geometric objects, such as triangles or points). Patches are also subjected to a Patch Cull test prior to tessellation. Culled patches are immediately discarded. The TE stage is entirely fixed function and does not spawn threads.

Programming Note
When MeshShading is Enabled, TE-related state is IGNORED by HW and the TE stage either (a) acts as a pass-though stage if TaskShader is disabled or (b) only participates in Task Redistribution if TaskShader is enabled (See TASK State description and 3DSTATE_TASK_REDISTRIB).

Patches are specified via URB handles output by the preceding Hull Shader stage. These handles reference Patch URB Entry data written into the URB by HS shaders. The tessellation process is controlled by TE state and Tessellation Factors (TFs) read from the Patch URB Entries.

The fixed-function tessellation algorithm is considered an implementation detail and is therefore beyond the scope of this document. That detail includes both the order of output topologies as well as the order of vertices (domain points) within the output topologies. Only a high-level overview is provided to describe how the (few) state variables can be used to control aspects of tessellation behavior. The implementation will generate deterministic results (given the same exact inputs it will produce exactly the same outputs).

Several domain types (QUAD, TRI, and ISOLINE) are supported. Depending on the domain type, the TE stage outputs the required point/line/triangle topologies including a domain point per vertex. These

topologies will be output to the DS stage, where the domain points will be converted to 3D object vertices, resulting in 3D objects as typically input to the 3D pipeline when HOS tessellation is not used.

When tessellation is disabled, all topologies (including patchlist topologies) simply pass through to the GS stage. When tessellation is enabled, only patchlist topologies should be issued to the pipeline, else behavior is UNDEFINED. The MI_TOPOLOGY_FILTER command can be used to ensure this happens, i.e., it can be used to have the Command Stream ignore 3DPRIMITIVE commands that do not match a specific topology type.

Enabling tessellation is accomplished by enabling the HS/TE/DS stages in specific combinations. Those valid combinations are described in the table below.

Valid Tessellation Enabled Configurations
To enable tessellation, the HS, TE, and DS stages must be enabled and disabled together. Other configurations will result in behavior that is UNDEFINED.

State

This section contains the state registers for the Tessellation Engine.

3DSTATE_TE

Programming Note
When MeshShading is Enabled, TE-related state is IGNORED by HW and the TE stage either (a) acts as a pass-through stage if TaskShader is disabled or (b) only participates in Task Redistribution if TaskShader is enabled (See TASK State description and 3DSTATE_TASK_REDISTRIB).

Functions

Patch Culling

Normally, if any "outside" TF is <= 0.0 or NaN, the entire patch is culled at the TE stage.

Inside TFs are not used to cull patches.

Tessellation Factor Limits

After the Patch Culling test is performed, the TessFactors undergo a min() clamp to either the **MaxTessFactorOdd** (for FRACTIONAL_ODD partitioning) or **MaxTessFactorNotOdd** (for FRACTIONAL_EVEN or INTEGER partitioning). Exception: If the ISOLINE domain is specified, the LineDensity TessFactor will be clamped to the **MaxFactorNotOdd** value even if FRACTIONAL_ODD partitioning is specified).

Partitioning

The Partitioning state controls how the TFs are used to divide their corresponding edges.

Partitioning Mode	Definition
INTEGER	The edge is divided into an integral number of equal segments (given some fixed-point tolerance). After clamping, the TF is rounded up to an integer value. The edge is divided into that many equal segments.
EVEN_FRACTIONAL	The edge is divided into an *even* number of possibly unequal segments. The total number of segments is determined by rounding up the post-clamped TF to an even number. More specifically, the edge is divided exactly in half. Like the endpoints of the edge, the midpoint of the edge is by definition a tessellation point. Each half contains some number of equal segments and possibly one smaller segment. The size of the smaller segment is determined by the position of the TF value within the range defined by the TF rounded down and up to even numbers. The closer the TF is to the smaller value, the smaller the segment size is. When the TF reaches the smaller even value, the smaller segment disappears. The closer the TF gets to the larger even value, the closer the smaller segment size approaches the size of the other segments. When the TF reaches the larger even value, all segments are equal. The position of the smaller segment along the half edge varies as a function of the TF value.
ODD_FRACTIONAL	The edge is divided into an *odd* number of possibly unequal segments. The tessellation scheme is very similar to EVEN_FRACTIONAL partitioning, except that the edge midpoint is not included as a tessellation point. This, and the fact that the tessellation points are mirrored about the edge midpoint, causes an "odd" segment (which may or may not be the "smaller" segment) to straddle the edge midpoint, therefore resulting in the number of segments for the edge always being odd.
POW2	The edge is divided into an integral number of equal segments (given some fixed-point tolerance). After clamping, the TF is rounded up to an integer power of 2 value. The edge is divided into that many equal segments.

Domain Types and Output Topologies

The major (if only) task of the TE stage is to tessellate a 2D (u,v) domain region, as selected by the Domain state, into some number of 2D object topologies. (If the patch is culled, that number may be zero). The options for Domain state are:

- QUAD: A square 2D region within a u,v Cartesian (rectangular) space. The region extends from the origin to u=1 and v=1. Within the region, tessellation domain locations are determined. The possible output topologies include points, clockwise triangles, and counter-clockwise triangles.

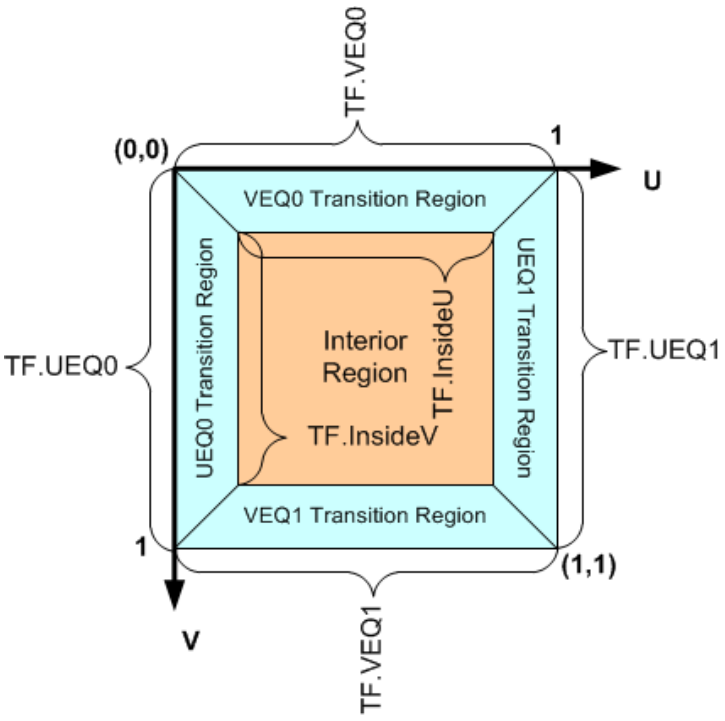
- TRI: A triangular 2D region with u,v,w barycentric (areal) coordinates. The three edges correspond to u=0, v=0, and w=0 boundaries. In barycentric coordinates, w = 1 - u - v, therefore points within the region are fully defined as 2D (u,v) coordinates. Within the region, tessellation domain locations are determined. The possible output topologies include points, clockwise triangles, and counter-clockwise triangles.
- ISOLINE: A series of points within a QUAD domain, where the points lie on lines parallel to the u axis and extending from [0,1) in the v direction. Either the segmented lines (linestrips) or individual point topologies can be output.

QUAD Domain Tessellation

The four "outside" TFs (TF.UEQ0, TF.VEQ0, TF.UEQ1, TF.VEQ1) are used to specify the level of tessellation along the four corresponding edges of the 2D quad domain. The two "inside" TFs (TF.InsideU, TF.InsideV) are used to determine the level of tessellation within a 2D "interior" region. Typically the interior region appears as a "regularly-tessellated 2D grid", however under certain conditions the interior region may collapse in which case only the outside TFs are relevant.

In general, a transition region exists between each edge of the interior region and the corresponding outside edge. The topologies generated for these regions effectively "stitch together" locations along the outside and inside edges, as each of these edges can contain a different number of tessellated segments. In the case where all TFs in a given direction (e.g., TF.VEQ0, TF.InsideU, and TF.VEQ1) are the same value, it appears as if the regularly-tessellated interior region extends all the way to the outside edges. If this condition simultaneously exists for both u and v directions, the entire domain will appear to be tessellated into a regular grid, with no noticeable transition regions.

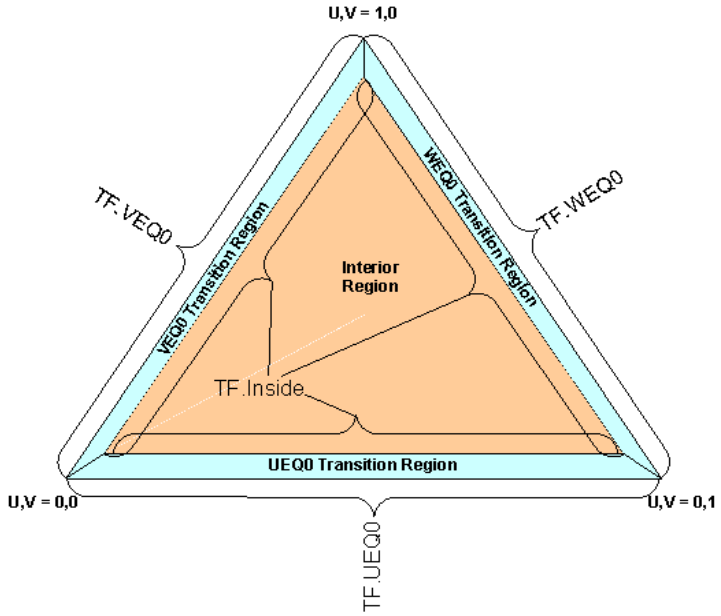
QUAD Domain



TRI Domain Tessellation

Tessellation of the TRI domain is similar to the QUAD domain, except only three outside edges/TFs are used, and the tessellation of the interior region is controlled by a single TF.

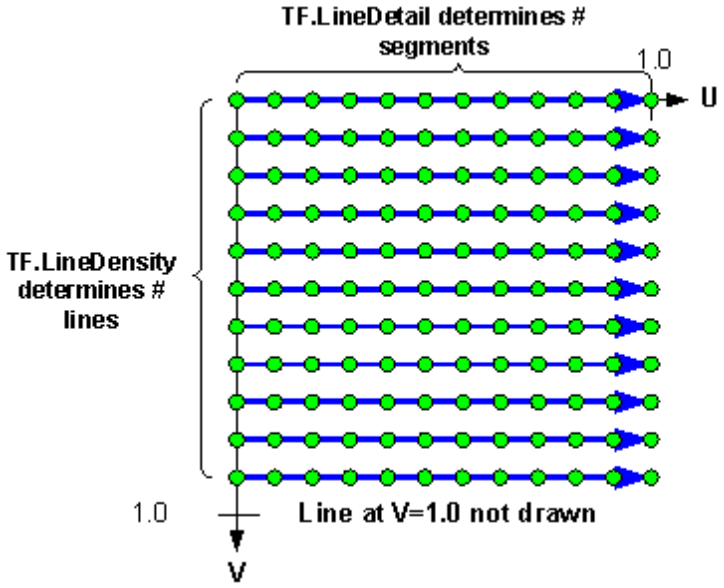
TRI Domain



ISOLINE Domain Tessellation

Tessellation of the ISOLINE domain is different but much simpler than QUAD and TRI domains. The TF.LineDetail TF controls how finely the U direction is tessellated, while the TF.LineDensity TF controls how finely the V direction is tessellated. When LINE output topology is selected, a series of segmented lines parallel to the U axis (constant V) are output. When POINT output topology is selected, only the line segment endpoints are output (as point objects). In either case there is no topology output for the V=1 edge, which avoids overlapping lines for adjacent patches.

ISOLINE Domain



Tessellation Redistribution

When multiple geometry pipes are enabled, workloads employing a significant amount of tessellation (especially high tessellation factors) may suffer performance issues due to pre-rasterization crossbar (pre-RX) buffering limitations. If the number of post-setup objects generated from the tessellation of a distributed batch of input PATCHLIST_x objects exceed the pre-RX buffering provided for the geometry pipe, the geometry pipeline will start to "back up" while it waits for its turn to output its post-setup objects to the RX. When the geometry pipeline is subsequently allowed to output to the RX, all of the post-setup objects resulting from the batch of patches (BOP) need to be completely drained to the RX.

If the geometry pipeline has backed up, the rate at which the pre-buffer post-setup objects drain will typically be slower than the rate at which the buffered objects will drain. As the current pipeline drains the buffered objects and then more slowly drains the remaining pre-buffer objects into the RX, the other pipelines will become stalled. As the number of post-setup objects resulting from the batch of patches grows in relation to the buffer size, the situation approaches the condition where only one geometry pipeline operates at any given time. At the extreme, the device shall approach the throughput of a single geometry pipeline.

Tessellation Redistribution feature is available to help avoid this situation. When enabled, HS-shaded patches are redistributed across the multiple pipelines at the Tessellation stage, as it is not until the Tessellation stage that the patch Tessellation Factors (and therefore the number of triangles generated by a patch) are known to the geometry pipeline. The redistribution can occur at the individual patch level or partial patch (patch region) level, significantly reducing the worst-case number of post-culled objects generated versus those generated by a batch of patches, thus avoiding the performance glass-jaw mentioned above.

While redistributing tessellation work to comprehend the post-cull buffering limitation of a pipeline is important, the redistribution process does incur some amount of performance overhead, and therefore SW-visible controls are provided to help manage this tradeoff. These controls are comprised of the following state variables:

- **3DSTATE_TE::TessellationDistributionMode**: This state is used to enable/disable the feature.
- **3DSTATE_TE::TessellationDistributionLevel:** This state is used to select between PATCH and REGION distribution granularity.
- **3DSTATE_TE::SmallPatchThreshold:** This state is used to classify patches as "small" or "large" patches, where small patches are not subjected to redistribution (i.e., are always processed by the local pipe that received the associated batch of patches). As small patches are not redistributed, it is possible for all the patches in a BOP to reach this threshold, and therefore SW shall take the **3DSTATE_VFG::PatchBatchSize*** state variable programming into account in order to manage the worst case triangle counts resulting from a BOP of small patches.
- **3DSTATE_TE::TargetBlockSize:** This state specifies a triangle count used as the target patch distribution granularity. The HW will use this value to guide the subdivision of large patches into regions when REGION granularity is specified. The value programmed also serves as the unit value for the 3DSTATE_TE::LocalBOPAccumulatorThreshold state variable.
- **3DSTATE_TE::LocalBOPAccumulatorThreshold:** This state is used to specify the maximum number of tessellated triangles to be processed by the local pipeline before patch redistribution is to occur.

Impact of Triangle Culling

The state variables listed above are based on pre-culled triangle counts. While the Tessellation stage can compute the number of triangles generated by a patch, it has no knowledge of the amount of culling performed by the pre-RX Clipping and front-end Setup functions. While SW may be conservative and assume 0% triangle culling when programming these states, higher performance may be possible if SW is aware of (or can speculate about) the typical triangle cull rate for an application and use that rate to scale up the thresholds/target values such that the post-culled triangle counts take better advantage of pre-RX buffer capacities.

Domain Shader (DS) Stage

The DS stage is very similar to the VS stage in that it is responsible for dispatching EU threads to shade vertices and maintaining a cache (with reference counts) of the shaded vertex outputs of these threads. Major differences are as follows:

- The DS receives topologies with "domain points" instead of vertices. The only data specific to a domain point are its U,V coordinates. These coordinates (plus a default or computed W coordinate) are passed directly in the DS thread payload. There is no other vertex-specific "input vertex data".
- The concatenation of the domain point U,V coordinates (vs. a vertex index) is used as the cache tag.
- The cache is invalidated between patches.

The DS stage accepts state information via the inline 3DSTATE_DS command.

Programming Note
When MeshShading is Enabled, DS-related state is IGNORED by HW and the DS stage acts simply as a pass-through stage.

State

This section contains the state registers for the Domain Shader.

Register
3DSTATE_DS
3DSTATE_PUSH_CONSTANT_ALLOC_DS
3DSTATE_CONSTANT_DS
3DSTATE_CONSTANT(Body)
3DSTATE_BINDING_TABLE_POINTERS_DS
3DSTATE_SAMPLER_STATE_POINTERS_DS
3DSTATE_URB_DS

Programming Note
When MeshShading is Enabled, DS-related state is IGNORED by HW and the DS stage acts simply as a pass-through stage.

Functions

DUAL_PATCH Thread Execution

When Dispatch Mode is set to SIMD8_SINGLE_OR_DUAL_PATCH mode, both the KSP and DUAL_PATCH KSP kernels are enabled. The DS stage decides whether to spawn a SINGLE_PATCH (KSP) or DUAL_PATCH thread dynamically, based on the number of domain points associated with patches. (See Implementation Note below).

- The KSP kernel operates exactly like when SIMD8_SINGLE_PATCH mode is set. Up to 8 domain points for a single patch are processed by the DS thread, which operates in SIMD8 fashion.
- The DUAL KSP kernel uses a hybrid SIMD8 execution mode. The 8 execution channels are divided into 4 upper channels associated with Patch 1, and 4 lower channels associated with Patch 0. Patch data is passed in SIMD4x2 layout, with Patch 1 data (Primitive ID, pushed URB data) in the upper 4 channels, and Patch 0 data in the lower 4 channels. The kernel operates much like SIMD8_SINGLE_PATCH mode, though it needs to access the appropriate SIMD4 patch data.

Implementation Note: Kernel selection is as follows: If a patch requires more than 4 domain points to be shaded, SIMD8_SINGLE_PATCH threads are spawned until 4 or fewer domain points remain. These domain points (if any exist) are held pending until the next patch is received. Likewise, if a patch requires 4 or fewer total domain points, those domain points are held pending. In either case, if the subsequent

patch requires 4 or fewer domain points to be shaded, a SIMD8_DUAL_PATCH thread is spawned to shade both sets of 4 or fewer domain points. If the subsequent patch requires more than 4 domain points, the (4 or fewer) buffered domain points of the previous patch are shaded via a SIMD8_SINGLE_PATCH thread, and the cycle continues.

Statistics Gathering

The DS stage maintains the DS_INVOCATIONS statistics counter, which counts the number of incoming domain points, irrespective of cache hit/miss. Note that this is different than VS_INVOCATIONS, which counts shader invocations and therefore doesn't count cache hits.

Payloads

SIMD8 Payload

The following table describes the payload delivered to DS threads.

DS Thread Payload (SIMD8)

DWord	Bits	Description
R0.7	31	
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Buffer.** Specifies the index of the scratch buffer allocated to the stage. Format = SurfaceStateOffset[27:6]
	9:0	**FFTID.** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address. Format = DynamicStateOffset[31:5]

DWord	Bits	Description
	4	Reserved
	3:0	Reserved
R0.2	31:0	Reserved: delivered as zeros (reserved for message header fields)
R0.1	31:0	**PrimitiveID.** This is the 32-bit PrimitiveID value associated with the patch. It is common to all output domain points resulting from the tessellation of the patch. Format: U32 The contents of this field is UNDEFINED if the 3DSTATE_DS::PrimitiveIDNotRequired state bit is set.
R0.0	31:25	Reserved
	24:0	**Patch URB Offset.** This is the offset within the URB where the patch data is stored. Format: 64B-granular offset into the URB Bits 24:16 select the per-slice URB. Bits 15:0 are the 64B-granular offset into that slice's URB.
R1.7	31:0	**Domain Point 7 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.6	31:0	**Domain Point 6 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.5	31:0	**Domain Point 5 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.4	31:0	**Domain Point 4 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.3	31:0	**Domain Point 3 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.2	31:0	**Domain Point 2 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.1	31:0	**Domain Point 1 U Coordinate.** (See Domain Point 0 U Coordinate.)
R1.0	31:0	**Domain Point 0 U Coordinate.** U coordinate associated with Domain Point 0. Format: FLOAT32
R2.7	31:0	**Domain Point 7 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.6	31:0	**Domain Point 6 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.5	31:0	**Domain Point 5 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.4	31:0	**Domain Point 4 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.3	31:0	**Domain Point 3 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.2	31:0	**Domain Point 2 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.1	31:0	**Domain Point 1 V Coordinate.** (See Domain Point 0 V Coordinate.)
R2.0	31:0	**Domain Point 0 V Coordinate.** V coordinate associated with Domain Point 0. Format: FLOAT32
R3.7	31:0	**Domain Point 7 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.6	31:0	**Domain Point 6 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.5	31:0	**Domain Point 5 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.4	31:0	**Domain Point 4 W Coordinate.** (See Domain Point 0 W Coordinate.)

DWord	Bits	Description
R3.3	31:0	**Domain Point 3 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.2	31:0	**Domain Point 2 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.1	31:0	**Domain Point 1 W Coordinate.** (See Domain Point 0 W Coordinate.)
R3.0	31:0	**Domain Point 0 W Coordinate.** If Compute W Coordinate Enable is set, this field will receive the computed value (1 - U - V) for Domain Point 0. Otherwise it is passed as 0.0. Format: FLOAT32
R4.7	31:0	**Domain Point 7 URB Return Handle.** (See R4.0.)
R4.6	31:0	**Domain Point 6 URB Return Handle.** (See R4.0.)
R4.5	31:0	**Domain Point 5 URB Return Handle.** (See R4.0.)
R4.4	31:0	**Domain Point 4 URB Return Handle.** (See R4.0.)
R4.3	31:0	**Domain Point 3 URB Return Handle.** (See R4.0.)
R4.2	31:0	**Domain Point 2 URB Return Handle.** (See R4.0.)
R4.1	31:0	**Domain Point 1 URB Return Handle.** (See R4.0.)
R4.0	31:16	Reserved: MBZ
	15:0	**Domain Point 0 URB Return Handle.** This is the offset within the URB where domain point 0 is to be stored. Format: U16 64B-granular offset into the Local URB
[Varies] optional	255:0	**Constant Data (optional):** Please refer to the Push Constants chapter in the General Programming of Thread-Generating Stages section for more details on size and source of constant data.
Varies [Optional]	255:0	**Patch URB Data (optional).** Some amount of Patch Data (possible none) can be extracted from the URB and passed to the thread in this location in the payload. The amount of data provided is defined by the Patch URB Entry Read Length state (3DSTATE_DS).

DUAL_PATCH Payload

The following table describes the payload delivered to DS threads.

DUAL_PATCH DS Thread Payload (SIMD8)

DWord	Bits	Description
R0.7	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time.

DWord	Bits	Description
		Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Buffer.** Specifies the index of the scratch buffer allocated to the stage. Format = SurfaceStateOffset[27:6]
	9:0	**FFTID.** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the General State Base Address or Dynamic State Base Address. Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	Reserved
R0.2	31:0	Reserved: delivered as zeros (reserved for message header fields)
R0.1	31:27	Reserved
	26:16	**Description** Reserved
	15:0	**Patch 1 URB Offset.** This is the offset within the URB where the Patch 1 data is stored. Format: U16 64B-granular offset into the URB This is a Local URB offset.
R0.0	31:27	Reserved
	26:16	**Description** Reserved
	15:0	**Patch 0 URB Offset.** This is the offset within the URB where the Patch 0 data is stored. Format: U16 64B-granular offset into the URB This is a Local URB offset.
		The following "R1" GRF is not included in the payload if 3DSTATE_DS::PrimitiveIDNotRequired is set.

DWord	Bits	Description
R1.5-7	31:0	Reserved
R1.4	31:0	**Patch 1 PrimitiveID.** This is the 32-bit PrimitiveID value associated with Patch 1. Format: U32
R1.1-3	31:0	Reserved
R1.0	31:0	**Patch 0 PrimitiveID.** This is the 32-bit PrimitiveID value associated with Patch 0. Format: U32
R2.7	31:0	**Patch 1 Domain Point 3 U Coordinate.** (See R2.0.)
R2.6	31:0	**Patch 1 Domain Point 2 U Coordinate.** (See R2.0.)
R2.5	31:0	**Patch 1 Domain Point 1 U Coordinate.** (See R2.0.)
R2.4	31:0	**Patch 1 Domain Point 0 U Coordinate.** (See R2.0.)
R2.3	31:0	**Patch 0 Domain Point 3 U Coordinate.** (See R2.0.)
R2.2	31:0	**Patch 0 Domain Point 2 U Coordinate.** (See R2.0.)
R2.1	31:0	**Patch 0 Domain Point 1 U Coordinate.** (See R2.0.)
R2.0	31:0	**Patch 0 Domain Point 0 U Coordinate.** U coordinate associated with Domain Point 0 of Patch 0. Format: FLOAT32
R3.7	31:0	**Patch 1 Domain Point 3 V Coordinate.** (See R3.0.)
R3.6	31:0	**Patch 1 Domain Point 2 V Coordinate.** (See R3.0.)
R3.5	31:0	**Patch 1 Domain Point 1 V Coordinate.** (See R3.0.)
R3.4	31:0	**Patch 1 Domain Point 0 V Coordinate.** (See R3.0.)
R3.3	31:0	**Patch 0 Domain Point 3 V Coordinate.** (See R3.0.)
R3.2	31:0	**Patch 0 Domain Point 2 V Coordinate.** (See R3.0.)
R3.1	31:0	**Patch 0 Domain Point 1 V Coordinate.** (See R3.0.)
R3.0	31:0	**Patch 0 Domain Point 0 V Coordinate.** V coordinate associated with Domain Point 0. Format: FLOAT32
R4.7	31:0	**Patch 1 Domain Point 3 W Coordinate.** (See R4.0.)
R4.6	31:0	**Patch 1 Domain Point 2 W Coordinate.** (See R4.0.)
R4.5	31:0	**Patch 1 Domain Point 1 W Coordinate.** (See R4.0.)
R4.4	31:0	**Patch 1 Domain Point 0 W Coordinate.** (See R4.0.)
R4.3	31:0	**Patch 0 Domain Point 3 W Coordinate.** (See R4.0.)
R4.2	31:0	**Patch 0 Domain Point 2 W Coordinate.** (See R4.0.)
R4.1	31:0	**Patch 0 Domain Point 1 W Coordinate.** (See R4.0.)
R4.0	31:0	**Patch 0 Domain Point 0 W Coordinate.** If Compute W Coordinate Enable is set, this field will

DWord	Bits	Description
		receive the computed value (1 - U - V) for Domain Point 0. Otherwise it is passed as 0.0. Format: FLOAT32
R5.7	31:0	**Patch 1 Domain Point 3 URB Return Handle.** (See R5.0.)
R5.6	31:0	**Patch 1 Domain Point 2 URB Return Handle.** (See R5.0.)
R5.5	31:0	**Patch 1 Domain Point 1 URB Return Handle.** (See R5.0.)
R5.4	31:0	**Patch 1 Domain Point 0 URB Return Handle.** (See R5.0.)
R5.3	31:0	**Patch 0 Domain Point 3 URB Return Handle.** (See R5.0.)
R5.2	31:0	**Patch 0 Domain Point 2 URB Return Handle.** (See R5.0.)
R5.1	31:0	**Patch 0 Domain Point 1 URB Return Handle.** (See R5.0.)
R5.0	31:16	Reserved
	15:0	**Patch 0 Domain Point 0 URB Return Handle.** This is the offset within the URB where Patch 0 Domain Point 0 is to be stored. Format: U16 64B-granular offset into the URB This is a Local URB offset.
[Varies] optional	255:0	**Constant Data (optional):** Please refer to the Push Constants chapter in the General Programming of Thread-Generating Stages section for more details on size and source of constant data.
		Patch 0,1 URB Data follows (optional). This data is read from the URB and pushed in the payload. The amount of data provided for each patch (which may be 0) is defined by the Patch URB Entry Read Length state (3DSTATE_DS). The data is read from the URB starting at the Patch URB Entry Read Offset into each patch, so leading data with the Patch URB entries can be skipped over. Patch 1 data is passed in the upper 128 bits, while Patch 0 data is passed in the lower 128 bits. This is similar to how URB data is pushed into SIMD4x2 kernels (VS, GS, etc.).
[Varies] optional	255:128	**Patch 1 URB Data.**
	127:0	**Patch 0 URB Data.**

Geometry Shader (GS) Stage

GS Stage Overview

The GS stage of the 3D Pipeline converts objects within incoming primitives into new primitives through use of a spawned thread. When enabled, the GS unit buffers incoming vertices, assembles the vertices of each individual object within the primitives, and passes those object vertices (along with other data) to the graphics subsystem for processing by a GS thread.

When the GS stage is disabled, vertices flow through the unit unmodified.

Programming Note
When MeshShading is Enabled, only MESH-related state is used by HW and all GS-related state is IGNORED by HW.

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Pipeline* chapter for a general description of a 3D Pipeline stage, as much of the GS stage operation and control falls under these "common" functions. I.e., most stage state variables and GS thread payload parameters are described in *3D Pipeline*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the GS stage, and any exceptions the GS stage exhibits with respect to common FF unit functions.

State

This sections contains the state registers for the Geometry Shader.

Registers
3DSTATE_GS (The state used by GS is defined with this inline state packet.)
3DSTATE_CONSTANT_GS
3DSTATE_CONSTANT(Body)
3DSTATE_PUSH_CONSTANT_ALLOC_GS
3DSTATE_BINDING_TABLE_POINTERS_GS
3DSTATE_SAMPLER_STATE_POINTERS_GS
3DSTATE_URB_GS

Programming Note
When MeshShading is Enabled, only MESH-related state is used by HW and all GS-related state is IGNORED by HW.

Functions

Object Staging

The GS unit's Object Staging Buffer (OSB) accepts primitive topologies as a stream of incoming vertices and spawns a thread for each individual object within the topology.

Thread Request Generation

Object Vertex Ordering

The following table defines the number and order of object vertices passed in the Vertex Data portion of the GS thread payload, assuming an input topology with *N* vertices. The ObjectType passed to the thread is, by default, the incoming PrimTopologyType. Exceptions to this rule (for the TRISTRIP variants) are called out.

The following table also shows which vertex is selected to provide PrimitiveID (bold, underlined vertex number). In general, the vertex selected is the last vertex for non-adjacent prims, and the next-to-last vertex for adjacent prims. Note, however, that there are exceptions:

- reorder-enabled TRISTRIP[_REV], TRISTRIP_ADJ
- "odd-numbered" objects in TRISTRIP_ADJ

PrimTopologyType	Order of Vertices in Payload	
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	GS Notes
POINTLIST	[0] = (0); [1] = (1); ...; [N-2] = (N-2);	
POINTLIST_BF	N/A	
LINELIST (N is multiple of 2)	[0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1)	
LINELIST_ADJ (N is multiple of 4)	[0] = (0,1,2,3); [1] = (4,5,6,7); ...; [(N/4)-1)] = (N-4,N-3,N-2,N-1)	

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices in Payload [<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	GS Notes
LINESTRIP (N >= 2)	[0] = (0,1); [1] = (1,2); ...; [N-2] = (N-2,N-1)	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT (N >= 4)	[0] = (0,1,2,3); [1] = (1,2,3,4); ...; [N-4] = (N-4,N-3,N-2,N-1)	LINESTRIP_ADJ_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
LINESTRIP_BF	N/A	
LINESTRIP_CONT	Same as LINESTRIP	Handled same as LINESTRIP
LINESTRIP_CONT_BF	N/A	
LINELOOP (N >= 2)	[0] = (0,1); [1] = (1,2); [N] = (N-1,0);	Not supported after GS.
TRILIST (N is multiple of 3)	[0] = (0,1,2); [1] = (3,4,5); ...; [(N/3)-1] = (N-3,N-2,N-1)	
RECTLIST, RECTLIST_SUBPIXEL	Same as TRILIST	Handled same as TRILIST
TRILIST_ADJ (N is multiple of 6)	[0] = (0,1,2,3,4,5); [1] = (6,7,8,9,10,11); ...; [(N/6)-1] = (N-6,N-5,N-4,N-3,N-2,N-1)	
TRISTRIP (Reorder Leading) (N >= 3)	[0] = (0,1,2); {TRISTRIP} [1] = (1,3,2); {TRISTRIP_REV} [k even] = (k,k+1,k+2) {TRISTRIP} [k odd] = (k,k+2,k+1) {TRISTRIP_REV}	"Odd" triangles have vertices reordered and identified as TRISTRIP to inform the thread.

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices in Payload [<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	GS Notes
	[N-3] = (see above)	
TRISTRIP (Reorder Trailing) (N >= 3)	[0] = (0,1,2) {TRISTRIP} [1] = (2,1,3) {TRISTRIP_REV}; ... [k even] = (k,k+1,k+2) {TRISTRIP} [k odd] = (k+1,k,k+2) {TRISTRIP_REV} [N-3] = (see above)	"Odd" triangles have vertices reordered and identified as TRISTRIP_REV to inform the thread.
TRISTRIP_REV (Reorder Leading) (N >= 3)	[0] = (0,2,1) {TRISTRIP_REV}; [1] = (1,2,3) {TRISTRIP}; ...; [k even] = (k,k+2,k+1) {TRISTRIP_REV} [k odd] = (k,k+1,k+2) {TRISTRIP} [N-3] = (see above)	"Even" triangles have vertices reordered and identified as TRISTRIP to inform the thread.
TRISTRIP_REV (Reorder Trailing) (N >= 3)	[0] = (1,0,2) {TRISTRIP_REV} [1] = (1,2,3) {TRISTRIP}; ...; [k even] = (k+1,k,k+2,) {TRISTRIP_REV} [k odd] = (k,k+1,k+2) {TRISTRIP} [N-3] = (see above)	"Even" triangles have vertices reordered and identified as TRISTRIP_REV to inform the thread.
TRISTRIP_ADJ (Reorder Leading) (N >= 6)	N = 6 or 7: [0] = (0,1,2,5,4,3) N = 8 or 9: [0] = (0,1,2,6,4,3);	Objects have vertices reordered.

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices in Payload [<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	GS Notes
	[1] = (2,5,6,7,4,0); ...; N >= 10: [0] = (0,1,2,6,4,3); [1] = (2,5,6,8,4,0); ...; [k>1, even] = (2k,2k-2, 2k+2, 2k+6,2k+4, 2k+3); [k>2, odd] = (2k, 2k+3, 2k+4, 2k+6, 2k+2, 2k-2);...; Trailing object: [(N/2)-3, even] = (N-6,N-8,N-4,N-1,N-2,N-3); [(N/2)-3, odd] = (N-6,N-3,N-2,N-1,N-4,N-8);	
TRISTRIP_ADJ (Reorder Trailing) (N >= 6)	N = 6 or 7: [0] = (0,1,2,5,4,3) N = 8 or 9: [0] = (0,1,2,6,4,3); [1] = (4,0,2,5,6,7); ...; N >= 10: [0] = (0,1,2,6,4,3); [1] = (4,0,2,5,6,8); ...; [k>1, even] = (2k,2k-2, 2k+2, 2k+6,2k+4, 2k+3); [k>2, odd] = (2k+2, 2k-2, 2k, 2k+3, 2k+4, 2k+6);...; Trailing object: [(N/2)-3, even] = (N-6,N-8,N-4,N-1,N-2,N-3); [(N/2)-3, odd] = (N-4,N-8,N-6,N-3,N-2,N-1);	OpenGL ordering rules (last non-adjacent vertex is the last - aka provoking - vertex of the triangle). Even triangles have the same ordering as Leading Vertex, odd triangle ordering is different (rotated 2 vertices).

PrimTopologyType	Order of Vertices in Payload	
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	GS Notes
TRIFAN (N > 2)	[0] = (0,1,2); [1] = (0,2,3); ...; [N-3] = (0, N-2, N-1);	
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON, POLYGON_CONT	Same as TRIFAN	POLYGON_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
QUADLIST	[0] = (0,1,2,3); [1] = (4,5,6,7); ...; [(N/4)-1] = (N-4,N-3,N-2,N-1);	Not supported after GS. QUADLIST primitives are converted into POLYGONS in VF, and therefore never reach the GS.
QUADSTRIP	[0] = (0,1,3,2); [1] = (2,3,5,4); ... ; [(N/2)-2] = (N-4,N-3,N-1,N-2);	Not supported after GS. QUADSTRIP primitives are converted into POLYGONS in VF, and therefore never reach the GS.

PrimTopologyType	Order of Vertices in Payload
PATCHLIST_1 PATCHLIST_2 PATCHLIST_3..32	[0] = (0); [1] = (1); ...; [N-2] = (N-2); [0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1) similar to above

Thread Execution

A GS thread is capable of performing arbitrary algorithms given the thread payload (especially vertex) data and associated data structures (binding tables, sampler state, etc.) as input. Output can take the form of vertices output to the FF pipeline (at the GS unit) or data written to memory buffers (UAVs).

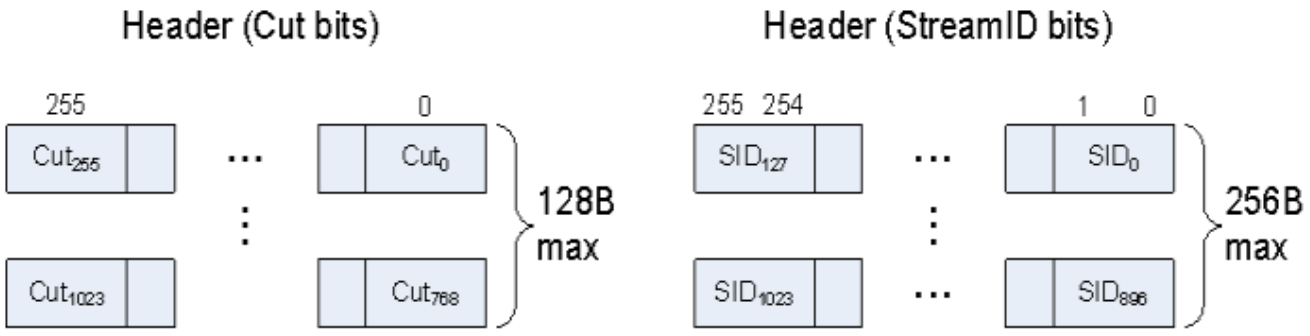
The primary usage models for GS threads include (possible combinations of):

- Compiled application-provided "GS shader" programs, specifying an algorithm to convert the vertices of an input object into some output primitives. For example, a GS shader may convert lines of a line strip into polygons representing a corresponding segment of a blade of grass centered on the line. Or it could use adjacency information to detect silhouette edges of triangles and output polygons extruding out from the those edges. Or it could output absolutely nothing, effectively terminating the pipeline at the GS stage.
- Driver-generated instructions used to write pre-clipped vertices into memory buffers (see Stream Output below). This may be required whether or not an app-provided GS shader is enabled.
- Driver-generated instructions used to emulate API functions not supported by specialized hardware. These functions might include (but are not limited to):
    - Conversion of API-defined topologies into topologies that can be rendered (e.g., LINELOOP, LINESTRIP, POLYGON, TRIFAN, QUADs, TRIFAN, etc.)
    - Emulation of "Polygon Fill Mode", where incoming polygons can be converted to points, lines (wireframe), or solid objects.
    - Emulation of wide/sprite points.

Thread Execution

GS URB Entry

All outputs of a GS thread are stored in the single GS thread output URB entry. Cut (1 bit/vertex) or StreamID (2 bits/vertex) bits are packed into an optional 1-8 32B header. The **Control Data Format** and **Control Data Header Size** states specify the size and contents of the header data (if any).



Following the optional header is a variable number of 16B or 32B-aligned/granular vertices:

- When rendering is DISABLED, typically output vertices are 32B-aligned, with the exception of 16B-alignment for vertices <= 16B in length.
    - The absolute worst-case size comes from three DW scalars output per vertex. If these are, say, three ".x" outputs, you need to store each DW in a 128b (16B) element, plus another pad 16B to keep the 32B alignment. So you require 4*16B = 64B/vertex. You have to have room for 1024 scalars / 3 scalar/vtx = 341 vertices. 341*64B = 21,824B. Then add 96B to hold 2b/vtx streamID and you get 21,920B entries.
- When rendering is ENABLED, each output vertex is 32B-aligned. Here the vertex header and vertex 'position' are required and therefore the minimum size vertex is 32B.
    - Here the worst-case size isn't as bad as render-disabled, as you have to have a 4DW position output, plus any additional output. So, say you output 5 DW per vertex. You need 64B/vertex (16B vtx header, 16B position, 16B for the 2nd element, and 16B of pad). You have to have room for 1024 scalars / 5 = 204 vertices. 204*64 = 13,056B. Then add 64B to hold 2b/vtx streamID and you get 13,120B entries.

The size of the URB entry should be based on the declared maximum # of output vertices and the declared output vertex size (the union of per-stream vertex structures, if required).

GS URB Entry - Output Vertex Count

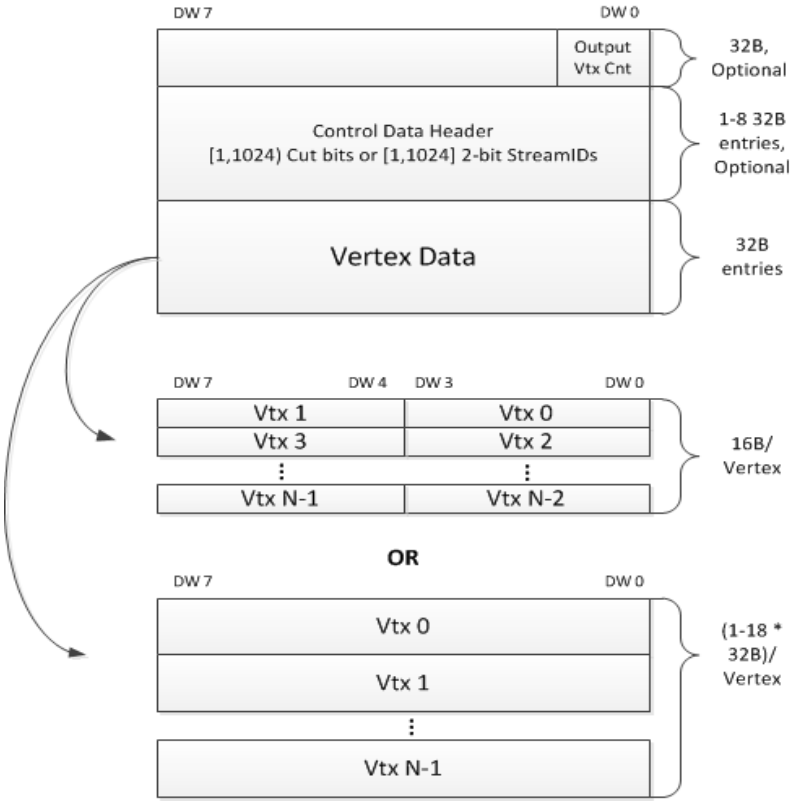
The GS URB entry is the same as in the two previous generations with the following exception: If **Static Output** (3DSTATE_GS) is clear, the URB entry starts with a 32B OUTPUT_VERTEX_COUNT structure as defined below. The control header (if present) immediately follows this structure. If **Static Output** is set, the control header (if present) appears at the very start of the URB entry (as described above).

GS OUTPUT_VERTEX_COUNT

DWord	Bit	Description
7:6	31:0	Reserved
0	31:16	Reserved
	15:0	**Output Vertex Count.** Indicates the number of vertices output from this GS shader invocation. Format = U16 Range: [0:1024]

This structure (if present) increases the maximum URB entry sizes (described above) by 32B.

The following diagram illustrates the possible layouts of a GS URB Entry:



GS Output Topologies

The following table lists which primitive topology types are valid for output by a GS thread.

PrimTopologyType	Supported for GS Thread Output?
LINELIST	Yes
LINELIST_ADJ	No
LINESTRIP	Yes
LINESTRIP_ADJ	No
LINESTRIP_BF	No
LINESTRIP_CONT	Yes
LINESTRIP_CONT_BF	No
LINELOOP	No
POINTLIST	Yes
POINTLIST_BF	No
POLYGON	Yes
QUADLIST	No
QUADSTRIP	No
RECTLIST	Yes
TRIFAN	Yes
TRIFAN_NOSTIPPLE	Yes

PrimTopologyType	Supported for GS Thread Output?
TRILIST	Yes
TRILIST_ADJ	No
TRISTRIP	Yes
TRISTRIP_ADJ	No
TRISTRIP_REV	Yes
PATCHLIST_xxx	Yes

GS Output StreamID

When the **GS Enable** is DISABLED, output vertices are assigned a StreamID = 0;

When the **GS Enable** is ENABLED, output vertices are assigned a StreamID = **Default StreamID** under the following conditions:

- **Control Data Header Size** = 0, or
- **Control Data Header Size** > 0 and **Control Data Format** = GSCTL_CUT

When the GS is enabled, **Control Data Header Size** > 0 and **Control Data Format** = GSCTL_SID, output vertices are assigned a StreamID as programmed in the Control Data output by the thread.

Primitive Output

(This section refers to output from the GS unit to the pipeline, not output from the GS thread)

The GS unit will output primitives (either passed-through or generated by a GS thread) in the proper order. This includes the buffering of a concurrent GS thread's output until the preceding GS thread terminates. Note that the requirement to buffer subsequent GS thread output until the preceding GS thread terminates has ramifications on determining the number of VUEs allocated to the GS unit and the number of concurrent GS threads allowed.

Statistics Gathering

There are a number of GS pipeline statistics counters associated with the GS stage and GS threads. This subsection describes these counters and controls depending on device, even in the cases where functions outside of the GS stage (e.g., DataPort) are involved in the statistics gathering.

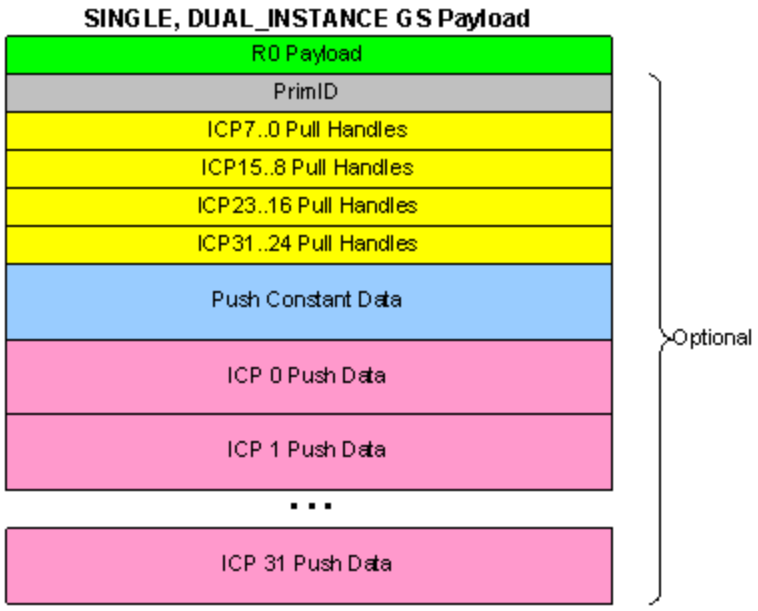
Refer to the *Statistics Gathering* summary provided earlier in this specification. Refer to the *Memory Interface Registers* chapter for details on these MMIO pipeline statistics counter registers, as well as the chapters corresponding to the other functions involved (e.g., DataPort, URB shared functions).

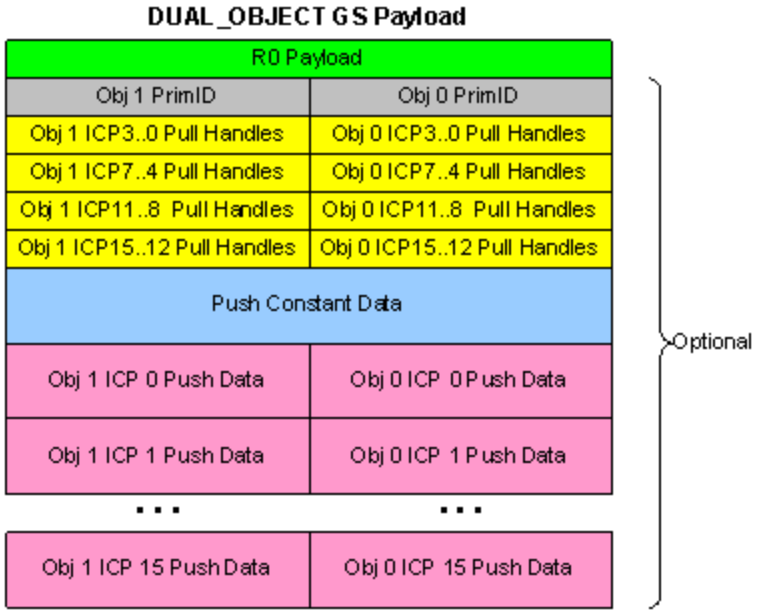
Payloads

Thread Payload High-Level Layout

GS Dispatch Layouts shows the high-level layout of the payload delivered to GS threads.

GS Dispatch Layouts





Subsequent sections provide detailed layouts for different processor projects.

SIMD8 Thread Payload

The table below shows the layout of the payload delivered to GS threads.

GRF DWord	Bits	Description
R0.7	31	
	30:0	Reserved.
R0.6	31	**Early Dereference Enable**: The enabe of early dereference. Reserved for Implementation Use
	30:24	Reserved.
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Buffer.** Specifies the index of the scratch buffer allocated to the stage. Format = SurfaceStateOffset[27:6]
	9:0	**FFTID.** This ID is assigned by the fixed function unit and is relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the Surface State Base Address. Format = SurfaceStateOffset[31:5]
	4:0	Reserved.
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the Dynamic State Base Address. Format = DynamicStateOffset[31:5]
	4	Reserved.
	3:0	Reserved
R0.2	31:23	Reserved.
	22	**Hint:** This is a copy of the corresponding 3DSTATE_GS bit. Format: U1
	21:16	**Primitive Topology Type.** This field identifies the Primitive Topology Type associated with the primitive containing this object. It indirectly specifies the number of input vertices included in the thread payload. Note that the GS unit may toggle this value between TRISTRIP and TRISTRIP_REV.

GRF DWord	Bits	Description
		If the **Discard Adjacency** bit is set, the topology type passed in the payload is UNDEFINED. Format: See 3D Pipeline
	15:0	Reserved.
R0.1	31:0	Reserved.
R0.0	31:16	Reserved.
	15:6	**Early Dereference ID**: The ID of the thread used for early dereference. Reserved for Implementation Use
	5:0	Reserved.
R1.7	31:0	GS Instance ID / URB Return Handle for Object 7 (See R1.0)
R1.6	31:0	GS Instance ID / URB Return Handle for Object 6 (See R1.0)
R1.5	31:0	GS Instance ID / URB Return Handle for Object 5 (See R1.0)
R1.4	31:0	GS Instance ID / URB Return Handle for Object 4 (See R1.0)
R1.3	31:0	GS Instance ID / URB Return Handle for Object 3 (See R1.0)
R1.2	31:0	GS Instance ID / URB Return Handle for Object 2 (See R1.0)
R1.1	31:0	GS Instance ID / URB Return Handle for Object 1 (See R1.0)
R1.0	31:27	**GS Instance ID 0.** For each input object, the GS unit can spawn multiple threads (instances). This field starts at zero for the first instance of an object and increments for subsequent instances. Format: U5
	26:16	Reserved.
	15:0	**URB Return Handle 0.** This is the URB offset where the EU's lower channels (DWords 3:0) results are to be stored. Format: U16 64B-aligned URB Offset This is an offset into the Local URB.
The following register is included only if **Include PrimitiveID** is enabled.		
R2.7	31:0	**Primitive ID 7.** This field contains the Primitive ID associated with input object 7 (or the single input object if InstanceCount>1) Format: U32
R2.6	31:0	**Primitive ID 6.** This field contains the Primitive ID associated with input object 6 (or the single input object if InstanceCount>1) Format: U32
R2.5	31:0	**Primitive ID 5.** This field contains the Primitive ID associated with input object 5 (or the single input object if InstanceCount>1)

GRF DWord	Bits	Description
		Format: U32
R2.4	31:0	**Primitive ID 4.** This field contains the Primitive ID associated with input object 4 (or the single input object if InstanceCount>1) Format: U32
R2.3	31:0	**Primitive ID 3.** This field contains the Primitive ID associated with input object 3 (or the single input object if InstanceCount>1) Format: U32
R2.2	31:0	**Primitive ID 2.** This field contains the Primitive ID associated with input object 2 (or the single input object if InstanceCount>1) Format: U32
R2.1	31:0	**Primitive ID 1.** This field contains the Primitive ID associated with input object 1 (or the single input object if InstanceCount>1) Format: U32
R2.0	31:0	**Primitive ID 0.** This field contains the Primitive ID associated with input object 0 (or the single input object if InstanceCount>1) Format: U32
The following registers are included only if **Include Vertex Handles** is enabled and **InstanceCount** == 1.		
Rn.7	31:16	Reserved.
	15:0	Object 7 ICP 0 Handle
Rn.6	31:16	Reserved.
	15:0	Object 6 ICP 0 Handle
Rn.5	31:16	Reserved.
	15:0	Object 5 ICP 0 Handle
Rn.4	31:16	Reserved.
	15:0	Object 4 ICP 0 Handle
Rn.3	31:16	Reserved.
	15:0	Object 3 ICP 0 Handle
Rn.2	31:16	Reserved.
	15:0	Object 2 ICP 0 Handle
Rn.1	31:16	Reserved.
	15:0	Object 1 ICP 0 Handle
Rn.0	31:16	Reserved.
	15:0	Object 0 ICP 0 Handle

GRF DWord	Bits	Description
[Rn+1]	255:0	ICP 1 Handle for Objects 0-7
[Rn+2]	255:0	ICP 2 Handle for Objects 0-7
...		...
[Rn+32]	255:0	ICP 32 Handle for Objects 0-7
The following registers are included only if **Include Vertex Handles** is enabled and **InstanceCount** > 1.		
Rn.7	31:16	Reserved.
	15:0	ICP 7 Handle (if required)
Rn.6	31:16	Reserved.
	15:0	ICP 6 Handle (if required)
Rn.5	31:16	Reserved.
	15:0	ICP 5 Handle (if required)
Rn.4	31:16	Reserved.
	15:0	ICP 4 Handle (if required)
Rn.3	31:16	Reserved.
	15:0	ICP 3 Handle (if required)
Rn.2	31:16	Reserved.
	15:0	ICP 2 Handle (if required)
Rn.1	31:16	Reserved.
	15:0	ICP 1 Handle (if required)
Rn.0	31:16	Reserved.
	15:0	ICP 0 Handle
[Rn+1]	255:0	ICP 8-15 Handle
[Rn+2]	255:0	ICP 16-23 Handle
[Rn+3]	255:0	ICP 24-31 Handle
[Varies] optional	255:0	**Constant Data (optional):** Please refer to the Push Constants chapter in the General Programming of Thread-Generating Stages section for more details on size and source of constant data.
Varies		**Pushed Vertex Data (InstanceCount == 1 Case): (optional)** Input data for the 8 input objects is located here. Object 0 (starting with Vertex 0 of Object 0) data is passed in DW0 of these GRFs, and Object 7 data is passed in DW7. The first GRF contains Vertex 0 Element 0 Component 0 for all 8 objects, followed by components 1-3 in the three subsequent GRFs. This is followed by GRFs containing Vertex 0 Element 1 (if it exists), and so on, up to the number of Vertex 0 elements specified by **Vertex URB Read Length**. This is followed by the data for Vertex 1 for all objects (if it exists), and so on until all relevant vertices are passed. Note that the amount of data passed is limited by the number of GRFs supported by EUs. Software is responsible for comprehending this limit and resorting to the pull model as required.

GRF DWord	Bits	Description
Rv.7	31:0	**Object 7 Vertex 0 Element 0 Component 0**
Rv.6	31:0	**Object 6 Vertex 0 Element 0 Component 0**
Rv.5	31:0	**Object 5 Vertex 0 Element 0 Component 0**
Rv.4	31:0	**Object 4 Vertex 0 Element 0 Component 0**
Rv.3	31:0	**Object 3 Vertex 0 Element 0 Component 0**
Rv.2	31:0	**Object 2 Vertex 0 Element 0 Component 0**
Rv.1	31:0	**Object 1 Vertex 0 Element 0 Component 0**
Rv.0	31:0	**Object 0 Vertex 0 Element 0 Component 0**
Rv+1	31:0	**Object 0-7 Vertex 0 Element 0 Component 1**
...		and so on...
Varies		**Pushed Vertex Data (InstanceCount > 1 Case): (optional)** Input data for the single input object (shared across all instances) is located here. The pushed data for Vertex 0 immediately follows any pushed constant data. The pushed data for Vertex 1 immediately follows Vertex 0, and so on. There is no upper/lower swizzling of data.

Mesh Shader (MESH) Stage

The primary function of the Mesh Shader stage is to request EU thread group dispatches for a series of MeshShader ThreadGroups (MeshTGs). Requests for MeshTG invocations originate either (a) directly from a 3DMESH command when TaskShaderDisabled or (b) indirectly from the output of a preceding TaskTG. In the latter case, each MeshTG is provided with the handle of the TUE output by the initiating TaskTG, thereby passing TaskTG data to the spawned MeshTGs.

When RasterizationDisabled, no output URB handles are provided to the MeshTGs and no additional processing is performed by the pipeline after the MeshTGs EOT.

When RasterizationEnabled, each MeshTG will be provided with an output MeshShader URB Entry (MUE) handle in the payload of each EU thread of the MeshTG dispatch. The threads of the MeshTG shall ensure that any generated geometry data is written into the MUE prior to the TG EOT. Subsequent to MeshTG EOT, the MeshShader stage shall process any output primitives as specified by the MUE data. If a non-zero count of primitives are output, the MeshShader stage will pass these primitives down the pipeline using the same vertex-based topologies used by the GeometryFF pipeline. One exception is that the MeshShader can output Per-Primitive Attributes in addition to Per-Vertex Attributes, where relevant CLIP and SBE state must be correctly programmed to support the Per-Primitive Attributes. The output primitives are passed down the pipeline in order of increasing TaskTG ID (if enabled) and MeshTG ID.

Mesh Shader State

This section contains links to the state commands for the Mesh Shader stage.

Register
3DSTATE_MESH_CONTROL
3DSTATE_MESH_DISTRIB
3DSTATE_MESH_SHADER
3DSTATE_MESH_SHADER_DATA
3DSTATE_URB_ALLOC_MESH

Programming Note
When MeshShading is Enabled, only MESH-related state is used by HW and all GS-related state is IGNORED by HW.

Mesh Shader Functions

This subsection provides details on the programming and behaviors of the Mesh Shader stage. See Mesh Shader Stage for an overview of these functions.

- In response to receipt of 3DMESH operations if TaskShaderDisabled or TaskShader requests for MeshShader dispatch if TasShaderEnabled:
  - If RasterizationEnabled, Mesh URB Entries are allocated in order to pass MUE output handles in MeshShader thread payloads
  - MeshShader thread requests are generated in order to initiate MeshTG dispatch and execution
  - MeshShader-specific statistics counters are incremented
  - If RasterizationEnabled, post-processing and topology generation is performed on (correctly-ordered) MUE data output by MeshTGs

Mesh Shader Thread Execution

Mesh Shader threads primarily execute as subsetted GPGPU Compute threads. Dispatch of Mesh Shader threads comes as a result of 3DMESH command execution or as requested by the MeshShaderLaunchCount output by am upstream TaskTG, both of which request the dispatch of a sequence of MeshShader ThreadGroups (MeshTGs).

Mesh Shader ThreadGroup Dispatch Dimension

ThreadGroup Dispatch Dimensions Supported
Only 1D TG dispatch is support by the 3DMESH command (if TaskShaderDisabled) and the TUE Header (if TaskShaderEnabled).
The 3DMESH command (and the corresponding TUE Header layout) supports either 1D and 3D (X,Y,Z) TG dispatch requests, as specified by 3DMESH::MeshDimensionSelect (1D|3D). A 3D TG dispatch request will, however, be converted to a 1D TG dispatch, with the count of TGs equal to the product of the X, Y and Z ThreadGroup Counts.

ThreadGroup Dispatch Dimensions Supported
The requested dispatch X,Y,Z dimensions from the 3DMESH command (or TUE Header) are passed into the MeshShader thread payload, and can be used by the kernel to generate 3D TGIDs from the 1D ThreadGroup ID also passed in the payload. This computation performed by the kernel can apply an arbitrary dispatch walk order (e.g., X-then-Y-then-Z, Z-then-X-then-Y, etc.).

The threads in a MeshTG can be provided with a BarrierID and SLM allocation, allowing GPGPU-like thread collaboration at a thread group level. Payload inputs to Mesh Shader threads are primarily a subset of GPGPU Compute thread payload inputs and include per-lane 1-D Local ID values used to identify an API thread (lane) within the local threadgroup. MeshShader payloads may also include an Indirect Parameter Pointer as well as one GRF of Inline Parameter Data.

The subsetting of GPGPU Compute capabilities include (but not limited to):

- MeshShader thread dispatches are 1D only.
- MeshShaders only utilize Bindless surfaces and samplers. Binding Table and Sampler State Pointers are not available/supplied.
- MeshShader thread dispatches do not support the following GPGPU Compute features:
  - Tiled walks
  - Partitioning
  - Mid-Thread Preemption

Unlike GPGPU Compute threads, MeshShader threads are provided with (1) if TaskShaderEnabled, an input Task URB Entry (TUE) Offset which points to per-TaskTG URB input data, and (2) if RasterizationEnabled, an output Mesh URB Entry (MUE) Offset which points to a per-MeshTG URB output allocation. The threads within a MeshTG are able to read the TUE data (if present) and collaborate in order to write the required output data into the MUE. The ability to pass this sort of temporary data via internal storage between different types of shaders is unique to Task-->Mesh Shaders and not available to GPGPU Compute threads (which can only pass outputs via memory). Mesh Shaders are also able to output via memory (especially if RasterizationDisabled), though like GPGPU Compute threads there is no HW-managed data passing (and allocation/deallocation bookkeeping) to subsequent threads.

Mesh URB Entry (MUE)

When RasterizationEnabled, Mesh Shader ThreadGroups (TGs) write shading results into a **Mesh Shader URB Entry (MUE)**. An MUE handle (offset into the Local URB) is passed in each thread's payload. The size of the MUE is specified by 3DSTATE_URB_ALLOC_MESH::MESHURBEntryAllocationSize. The handle will be 64B-aligned.

The threads typically collaborate to generate the shading results, but whether multiple threads or only one specific thread writes the results into the MUE is not dictated by hardware.

The MUE data is comprised of four components, of which only the first (NumPrims value) is required to be valid:

- **NumPrims**: This is a count of primitives output by the MeshShader TG. It is a DWord located at the very start of the MUE. The value programmed shall be less than or equal to the number of

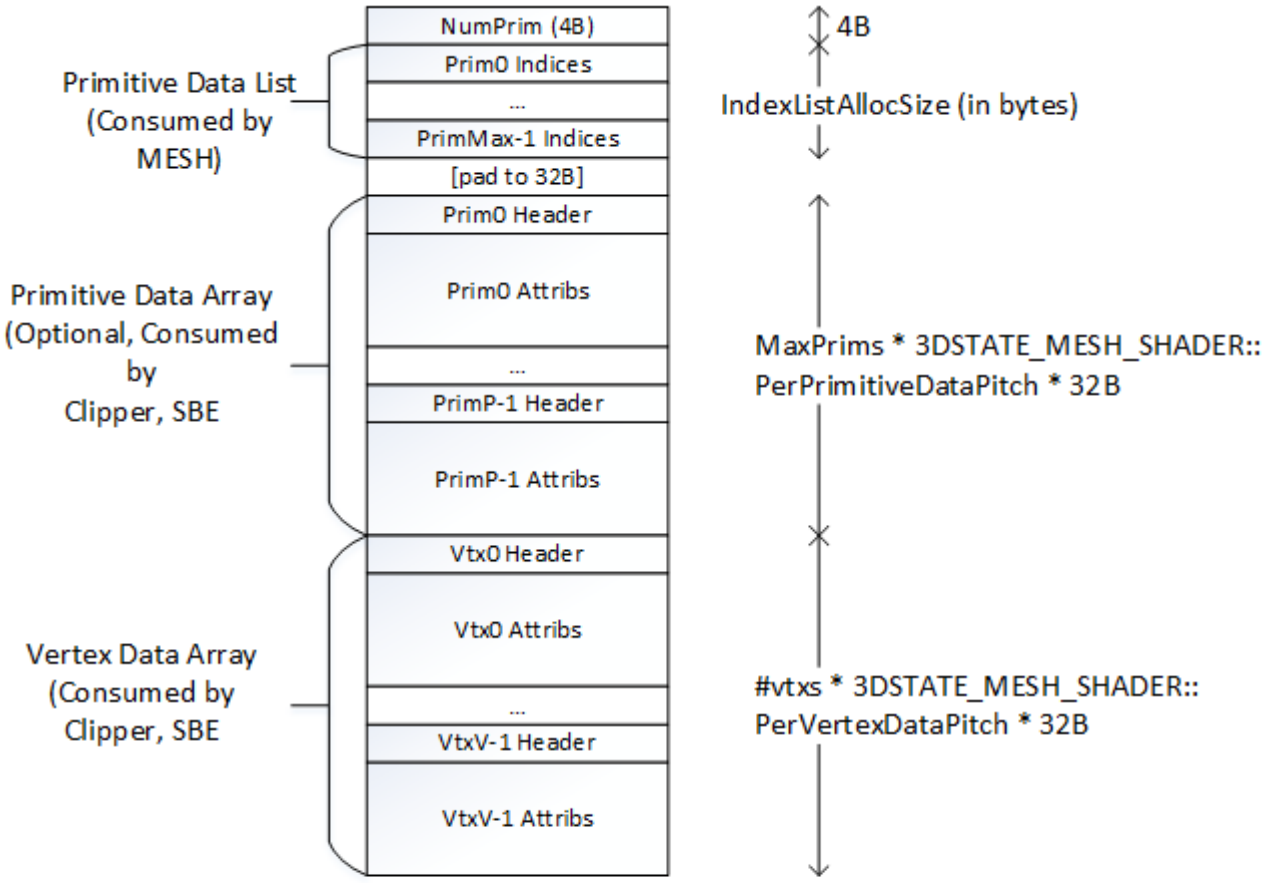
primitives specified in the 3DSTATE_MESH_SHADER::MaxPrimitiveCount state. (Note that that state is programmed as #Prims-1).

- A value of zero shall notify the HW that no primitives were generated. In this case the rest of the MUE is ignored by HW and the only subsequent action taken by HW for the MeshShader TG will be to free the MUE for reuse.
- A non-zero value shall instruct the HW to process the specified number of output primitives. Each primitive is defined by (a) a set of local vertex indices supplied in the Primitive Index List, and (b) a corresponding entry in the Primitive Data Array (if 3DSTATE_MESH_SHADER::PerPrimitiveDataPitch is non-zero). The local vertex indices are used to access corresponding entries in the Vertex Data Array. NumPrims shall not exceed the number of primitives specified by 3DSTATE_MESH_SHADER::MaxPrimitiveCount.

- **Primitive Index List**: The vertex data associated with each output primitive is specified by a set of local vertex indices contained within a set of consecutive bytes in the Primitive Index List.
  - The allocated size of the PrimitiveIndexList (in bytes) is specified by 3DSTATE_MESH_SHADER::MaxPrimitiveCount, 3DSTATE_MESH_SHADER::IndexFormat, and 3DSTATE_MESH_SHADER::OutputTopology. See the table below for how the allocated size (IndexListAllocSize) is determined by HW. The NumPrims value output by the MeshShaderTG determines how much of the allocated size contains valid primitive indices output by the MeshShaderTG.
  - The index values are "n"-bit, 0-based unsigned integer values. These indices identify entries in the Vertex Data Array, where multiple primitives and multiple indices within a primitive may specify the same index values (and therefore share vertex entries). The index values shall not reference a Vertex Data Array entry that starts or extends beyond the end of the MUE.
  - The layout of the index list is specified by 3DSTATE_MESH_SHADER::IndexFormat:
    - For packed formats (U888X, U101010X), each primitive consumes 32b in the index list, regardless of OutputTopology. OutputTopology will, however, specify which packed indices with those 32b is valid and used by HW. Use of these packed formats (if possible) may reduce the overall size of the PrimitiveIndexList, and may corresponding reduce the allocation size of the MUE, possibly allowing more MUEs (and/or TUEs) to be allocated, possibly resulting in higher performance.
    - For unpacked formats (U8, U16, U32), each primitive consumes a contiguous and ordered set of index values, without any padding between primitives. The size of each index is determined by the IndexFormat. The number of indices per primitive is indirectly specified by OutputTopology (POINT: 1 index, LINE: 2 indices, TRI: 3 indices).
- **Primitive Data Array**: The optional Primitive Data Array contains Per-Primitive attributes.
  - The Primitive Data Array starts on the first 32B-aligned offset following the Primitive Index List, therefore 0-31 bytes of padding may exist between the two structures.
  - The amount of Per-Primitive attribute storage reserved for each primitive is specified with 32B granularity by 3DSTATE_MESH_SHADER::PerPrimitiveDataPitch. If that state is set to zero, there shall be no Per-Primitive attributes and the Primitive Data Array shall not exist.
  - The array size (in primitives) is specified by 3DSTATE_MESH_SHADER::MaxPrimitiveCount.

- The format of the Per-Primitive attribute data is described in a subsection below.

- **Vertex Data Array:** The Vertex Data Array contains Per-Vertex attributes.
  - The Vertex Data Array shall be 32B-aligned. If there is no Primitive Data Array allocated, 1-7 DWords of padding may exist between the Primitive Index List and the Vertex Data Array. If a Primitive Data Array is allocated, the Vertex Data Array shall immediately follow the Primitive Data Array and be naturally 32B-aligned.
  - The amount of Per-Vertex attribute storage reserved for each vertex is specified with 32B granularity by 3DSTATE_MESH_SHADER::PerVertexDataPitch. There shall be at least 32B allocated per vertex.
  - The format of the Per-Vertex attribute data is described in a subsection below.



MaxPrims = 3DSTATE_MESH_SHADER::MaximumPrimitiveCount+1 = maxPrims decl'd value
P = 3DSTATE_MESH_SHADER::MaximumPrimitiveCount

The following table defines **IndexListAllocSize**, the maximum (allocated) size of the **Primitive Index List** (in bytes):

Index Format	POINT	LINE	TRI
U8	MaxPrims	MaxPrims*2	MaxPrims*3
U16	MaxPrims*2	MaxPrims*4	MaxPrims*6
U32	MaxPrims*4	MaxPrims*8	MaxPrims*12
U888X	MaxPrims*4	MaxPrims*4	MaxPrims*4
U101010X	MaxPrims*4	MaxPrims*4	MaxPrims*4

Per-Primitive Attribute Data

If 3DSTATE_MESH_SHADER::PerPrimitiveDataPitch is non-zero, there will be some number of 32B blocks allocated per primitive for Per-Primitive attributes.

The Per-Primitive data starts with an optional "**Primitive Header**" that contains Per-Primitive, System-Interpreted Values (SIVs) which are consumed (interpreted) by HW. The following table lists these Per-Primitive SIVs and their location in the Primitive Header (see Mesh Shader Vertex and Primitive Headers for more details on the MUE Headers):

Per-Primitive SIVs Located in the MUE Primitive Header

SIV Name	Primitive Header Location
Requested CPS Size X,Y	DWord 0
RenderTargetArray Index	DWord 1
Viewport Index	DWord 2
Cull Primitive Mask	DWord 3

If the MeshShader TGs are required to output one of more of the HW-consumable Per-Primitive values, SW shall perform the following actions:

- Set 3DSTATE_CLIP_MESH::PrimitiveHeaderEnable to ENABLE. This will instruct the Clip stage to read back and process the first 32B of the Primitive Data Array entries. The first 32B of the Per-Primitive Data Array entry (when PrimitiveHeaderEnable is set) is known as the "**Primitive Header**".
- Cause the kernel to write the required SIV value(s) into the Primitive Header.

If the MeshShader TGs are **not** required to output any of the HW-consumable Per-Primitive SIVs, SW shall set 3DSTATE_CLIP_MESH::PrimitiveHeaderEnable to DISABLE. In this case HW will default those HW-consumable values to **zero**.

The remainder of the Primitive Header DWs (if present), as well as all subsequent 32B blocks in the Per-Primitive Data Array entries, are ignored by the Clip stage and can be used for passing SW-defined Per-Primitive attributes to the Setup HW.

Per-Vertex Attribute Data

3DSTATE_MESH_SHADER::PerVertexDataPitch specifies the number of 32B blocks per vertex in the Per-Vertex Data Array. Each entry starts with a Vertex Header.

Mesh Shader Vertex and Primitive Headers

Software must ensure that any Per-Vertex data in the MUE starts with a valid Vertex Header and any, if any HW-consumable System-Interpreted Values (SIVs) are required as outputs, each primitive's Per-Primitive data in the MUE starts with a valid Primitive Header, both of which are described below.

The following table defines the MUE Vertex Header.

MUE Vertex Header

DWord	Bits	Description
D0	31:16	**Requested Coarse Pixel Size Y** (See *Vertex URB Entry (VUE) Formats*).
	15:0	**Requested Coarse Pixel Size X** (See *Vertex URB Entry (VUE) Formats*).
D0	31:0	Reserved
D1	31:0	Reserved
D2	31:0	Reserved
D3	31:0	**Point Width.** (See *Vertex URB Entry (VUE) Formats*).
D4	31:0	**Vertex Position 0 X Coordinate.** (See *Vertex URB Entry (VUE) Formats*).
D5	31:0	**Vertex Position 0 Y Coordinate.** (See *Vertex URB Entry (VUE) Formats*).
D6	31:0	**Vertex Position 0 Z Coordinate.** (See *Vertex URB Entry (VUE) Formats*).
D7	31:0	**Vertex Position 0 W Coordinate.** (See *Vertex URB Entry (VUE) Formats*).
If (N>0): D8 thru D[8+(4*(N-1))+3]	31:0	**Vertex Position [1].XYZW** through **Vertex Position [N].XYZW**. (See *Vertex URB Entry (VUE) Formats*).
D8	31:0	**ClipDistance 0 Value (optional).** (See *Vertex URB Entry (VUE) Formats*).
D9	31:0	**ClipDistance 1 Value (optional).** See above.
D10	31:0	**ClipDistance 2 Value (optional).** See above.
D11	31:0	**ClipDistance 3 Value (optional).** See above.

DWord	Bits	Description
D12	31:0	**ClipDistance 4 Value (optional).** See above.
D13	31:0	**ClipDistance 5 Value (optional).** See above.
D14	31:0	**ClipDistance 6 Value (optional).** See above.
D15	31:0	**ClipDistance 7 Value (optional).** See above.
	31:0	**End of Vertex Header Padding** (See *Vertex URB Entry (VUE) Formats*).
	31:0	**(Remainder of Vertex Elements).** (See *Vertex URB Entry (VUE) Formats*).

The following table defines the MUE Primitive Header.

MUE Primitive Header

DWord	Bits	Description
D0	31:16	**Requested Coarse Pixel Size Y** (See *Vertex URB Entry (VUE) Formats*).
	15:0	**Requested Coarse Pixel Size X** (See *Vertex URB Entry (VUE) Formats*).
D0	31:0	Reserved
D1	31:0	**Render Target Array Index (RTAIndex).** (See *Vertex URB Entry (VUE) Formats*).
D2	31:0	**Viewport Index.** (See *Vertex URB Entry (VUE) Formats*).
D3	31:0	Reserved
	31:16	Reserved
	15:0	**Cull Primitive Mask**. This field contains cull indication(s) for the primitive, as generated by the MeshShaderTG. If a bit is set, the corresponding primitive is to be culled (discarded). Otherwise the primitive is subject to further processing by the Clip, Setup stages, which may also lead to the primitive being culled (or not). Bit 0 corresponds to the "first" instantiation of the primitive, Bit 1 corresponds to the first replica of the primitive, Bit 2 corresponds to the second replica, and so on. See the Vertex Position fields in *Vertex URB Entry (VUE) Formats* for more information regarding Primitive Replication (e.g., how the number of valid replicas is defined/programmed).
D4	31:0	Reserved

DWord	Bits	Description
D5	31:0	Reserved
D6	31:0	Reserved
D7	31:0	Reserved
	31:0	**(Remainder of Per-Primitive Elements).** The **3DSTATE_SBE_MESH::Per-Primitive URB Entry Output Read Length** has a maximum value of 31 256-bit units. Of course, the actual allocated size of the MUE can and will limit the amount of data in an MUE.

Mesh Shader Statistics Gathering

The **MESH_INVOCATION_COUNT** MMIO register in each GSlice accumulates API-level Mesh Shader invocations dispatched by the pipeline. For each MeshShader ThreadGroup dispatched, this register is incremented by the thread group size. This register does not count EU thread dispatches.

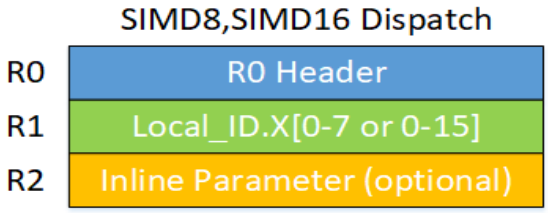
SW shall comprehend that a pipeline flush is required to ensure that preceding MeshShader work is included in the register value.

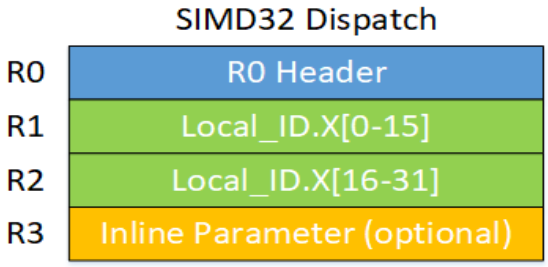
Mesh Shader Payloads

The MeshShader thread payload appears mostly as a subset of GPGPU Compute thread payloads, with the exception of the TaskShader and MeshShader URB Entry Offsets. Additional information can therefore be found in the relevant GPGPU Compute thread payload subsections.

As illustrated below, the MeshShader thread payload is comprised of two or three sections.

- **R0 Header** (contents specified below)
- Optional, Per-lane, 16-bit **Local_ID.X** values, passed in R1 for SIMD8 or SIMD16 dispatch or R1,R2 for SIMD32 dispatch. The value for a given lane identifies the local thread (lane) position within the API threadgroup (which may include multiple EU thread dispatches).
- Optional, 1-GRF **Inline Parameter**, included if 3DSTATE_MESH_SHADER::EmitInlineParameter is set. This will be passed immediately following the Local_ID section (in R2 for SIMD8 or SIMD16, or in R3,R4 for SIMD32). When enabled, SW shall provide the inline data via 3DSTATE_MESH_SHADER_DATA::InlineData state.





The **RO Header** of the Thread Dispatch Payload for a Mesh Shader thread:

DWord	Bits	Description
R0.7	31:0	**Task Shader URB Entry Offset:** If the Task Shader is enabled, this is the offset of the thread group's input Task Shader URB Entry (TUE) within the Slice's Local URB. Format: 64B-aligned Local URB Offset Bits 24:16 select the per-slice URB. Bits 15:0 are the 64B-granular offset into that slice's URB.
R0.6	31:16	Reserved: MBZ
	31:16	**DISP_MIN_X**: This field contains the Dispatch Dimension X value provided in the provoking 3DMESH command (if TaskShaderDisabled) or the provoking TUE Header (if TaskShaderEnabled) when 3DMESH::MeshDispatchDimension == 3D, Otherwise the contents of the field are UNDEFINED. Format: U16
	15:0	**Mesh Shader URB Entry Offset:** This is the offset of the thread group's output Mesh Shader URB Entry (MUE) within the Slice's Local URB. Format: U16 64B-aligned Local URB Offset
R0.5	31:10	**Scratch Space Buffer.** Specifies the surface state index to the Scratch Buffer for use by the kernel. This surface state index is relative to the **Surface State Base Address**. Format = SurfaceStateOffset[27:6]
	9:0	**FFTID.** This ID is assigned by HW and is a unique identifier for the thread in comparison to other concurrent threads (of any thread group). It is used to free up resources used by the thread upon thread completion. Format = U9

DWord	Bits	Description
R0.4	31:0	Reserved: MBZ
	31:16	**DISP_MIN_Z**: This field contains the Dispatch Dimension Z value provided in the provoking 3DMESH command (if TaskShaderDisabled) or the provoking TUE Header (if TaskShaderEnabled) when 3DMESH::MeshDispatchDimension == 3D, Otherwise the contents of the field are UNDEFINED. Format: U16
	15:0	**DISP_MIN_Y**: This field contains the Dispatch Dimension Y value provided in the provoking 3DMESH command (if TaskShaderDisabled) or the provoking TUE Header (if TaskShaderEnabled) when 3DMESH::MeshDispatchDimension == 3D, Otherwise the contents of the field are UNDEFINED. Format: U16
R0.3	31:0	Reserved: MBZ
	31:0	**Extended Parameter 0 (XP0)**: When 3DSTATE_MESH_SHADER::XP0Required is set, this field is a copy of the Extended Parameter 0 value from the provoking 3DMESH command (if 3DMESH::ExtendedParameter0Present was set) or zero (if 3DMESH::ExtendedParameter0Present was clear). If 3DSTATE_MESH_SHADER::XP0Required is clear, this field is UNDEFINED. Format U32
R0.2	31:24	**Number of Threads in GPGPU Thread Group.** This field specifies the number of EU threads that are in each Mesh Shader thread group.
	14:11	Reserved: MBZ
	9	Reserved: MBZ
	8	**Barrier Enable.** This field indicates that a barrier has been allocated for this kernel.
	7:0	**Thread Number Within Thread Group**. This field is used to identify a particular thread within a thread group. Format: U8
R0.1	31:0	**1D Thread Group ID:** This field identifies the X component of the thread group that this thread belongs to.
R0.0	31:5	**Indirect Data Start Address**: This field specifies the Graphics Memory starting address of the data to be loaded into the kernel for processing. This pointer is relative to the **General State Base Address.** It is the 64-byte aligned address of the indirect data. Format: GeneralStateOffset[31:6]
	4:1	Reserved: MBZ
	0	**Local ID Present:** Indicates that the HW-generated Local ID X follows the payload.

Stream Output Logic (SOL) Stage

The Stream Output Logic (SOL) stage receives 3D topologies originating in the VF, DS or GS stage. If enabled, the SOL stage uses programmed state information to copy portions of the vertex data associated with the incoming topologies across one or more Stream Output (SO) Buffers.

State

This section contains state commands and structures pertaining to the StreamOut Logic (SOL) stage of the 3D pipeline.

3DSTATE_STREAMOUT

The 3DSTATE_STREAMOUT command specifies control information for the SOL stage. Included are enables and sizes for input streams and enables for output buffers.

3DSTATE_STREAMOUT

3DSTATE_SO_DECL_LIST Command

The 3DSTATE_SO_DECL_LIST instruction defines a list of Stream Output (SO) declaration entries (SO_DECLs) and associated information for all specific SO streams in parallel.

3DSTATE_SO_DECL_LIST

SO_DECL

3DSTATE_SO_BUFFER

The 3DSTATE_SO_BUFFER command specifies the location and characteristics of an SO buffer in memory.

Command
3DSTATE_SO_BUFFER
3DSTATE_SO_BUFFER_INDEX_0
3DSTATE_SO_BUFFER_INDEX_1
3DSTATE_SO_BUFFER_INDEX_2
3DSTATE_SO_BUFFER_INDEX_3

The SOL Unit also receives 3DSTATE_INT which is transparent to SW. 3DSTATE_INT provides 3DSTATE_WM, 3DSTATE_PS_EXTRA, and 3DSTATE_DEPTH_STENCIL_STATE fields.

Signal	Description	Formula
SOL_INT::Render_Enable	If clear, the SO stage will not forward any topologies down the pipeline. If set, the SO stage will forward topologies associated with Render Stream Select down the pipeline. This bit is used even if SO	= (3DSTATE_STREAMOUT::Force_Rending == Force_On) || ((3DSTATE_STREAMOUT::Force_Rending != Force_Off) && !(3DSTATE_GS::Enable && 3DSTATE_GS::Output Vertex Size == 0) && !3DSTATE_STREAMOUT::API_Render_Disable && (3DSTATE_DEPTH_STENCIL_STATE::Stencil_TestEnable || 3DSTATE_DEPTH_STENCIL_STATE::Depth_TestEnable ||

Signal	Description	Formula
	Function Enable is DISABLED.	3DSTATE_DEPTH_STENCIL_STATE::Depth_WriteEnable || 3DSTATE_PS_EXTRA::PS_Valid || 3DSTATE_WM::Legacy Depth_Buffer_Clear || 3DSTATE_WM::Legacy Depth_Buffer_Resolve_Enable || 3DSTATE_WM::Legacy Hierarchical_Depth_Buffer_Resolve_Enable))

DW1[21]	DW1[20]	Stream Offset	Action
Full legacy mode. HW doesn't LOAD or STORE, it simply updates the MMIO register during stream out. SW can the LOAD/STORE using MI_LOAD_REG/ MI_STORE_REG.			
0	0	not equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = no action
0	0	equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = no action
SW can cause the LOAD of the SO_OFFSET using MI_LOAD_REG, and HW performs the STORE.			
0	1	not equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = No action, write SO_WRITE_OFFSET[x] to memory
0	1	equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = No action, write SO_WRITE_OFFSET[x] to memory
HW performs the LOAD, and SW can cause the STOREs using MI_STORE_REG_MEM.			
1	0	not equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = stream offset
1	0	equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = load from memory
HW performs both the LOAD and STORE.			
1	1	not equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = stream offset, write SO_WRITE_OFFSET[x] to memory
1	1	equal to 0xFFFFFFFF	SO_WRITE_OFFSET[x] = load from memory, write SO_WRITE_OFFSET[x] to memory

"SO_WRITE_OFFSET[x] =" occurs before the execution of the primitive, while write SO_WRITE_OFFSET[x] to memory occurs after the execution of the primitive.

Functions

Input Buffering

For the purposes of stream output, the SOL stage breaks incoming topologies into independent objects without adjacency information. In the process, any adjacent-only vertices are ignored. For example, it converts TRISTRIP_ADJ into independent 3-vertex triangles. However, if rendering is enabled, incoming topologies are passed to the Clip stage unmodified and therefore the Clip unit must be enabled if there is any possibility of "ADJ" topologies reaching it.

Note that the SOL unit will not see incomplete objects: the VF will remove incomplete input objects, the GS will remove GS-generated incomplete objects, and the DS does not output incomplete objects as only complete topologies are generated by the TE stage.

The OSB (Object Staging Buffer) reorders the vertices of odd-numbered triangles in TRISTRIP topologies to match API requirements.

Incoming topologies are tagged with a 2-bit StreamID. The StreamID is 0 for topologies originating from the VF stage (i.e., 3DPRIMITIVE_xxx). For topologies output from the GS stage, the StreamID is set by the GS shader. A Stream *n* Vertex Length is associated with each stream and defines how much data is read from the URB for vertices in that stream.

The following table specifies how the SOL stage streams out object vertices for each incoming topology type.

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices Streamed Out [<object#>] = (<vert#>,...);	Any SOL Notes
POINTLIST POINTLIST_BF	[0] = (0); [1] = (1); ...; [N-2] = (N-2);	
LINELIST (N is multiple of 2)	[0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1)	
LINELIST_ADJ (N is multiple of 4)	[0] = (1,2); [1] = (5,6); ...; [(N/4)-1)] = (N-3,N-2)	
LINESTRIP LINESTRIP_BF LINESTRIP_CONT LINESTRIP_CONT_BF (N >= 2)	[0] = (0,1); [1] = (1,2); ...; [N-2] = (N-2,N-1)	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT (N >= 4)	[0] = (1,2); [1] = (2,3); ...; [N-4] = (N-3,N-2)	LINESTRIP_ADJ_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
LINELOOP	N/A	Not supported after VF.
TRILIST (N is multiple of 3)	[0] = (0,1,2); [1] = (3,4,5); ...; [(N/3)-1] = (N-3,N-2,N-1)	
RECTLIST, RECTLIST_SUBPIXEL	Same as TRILIST	Handled same as TRILIST.
TRILIST_ADJ (N is multiple of 6)	[0] = (0,2,4); [1] = (6,8,10); ...; [(N/6)-1] = (N-6,N-	

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices Streamed Out [<object#>] = (<vert#>,...);	Any SOL Notes
	4,N-2)	
TRISTRIP (N >= 3) REORDER_LEADING	[0] = (0,1,2); [1] = (1,3,2); [k even] = (k,k+1,k+2) [k odd] = (k,k+2,k+1) [N-3] = (see above)	"Odd" triangles have vertices reordered to yield increasing leading vertices starting with v0.
TRISTRIP (N >= 3) REORDER_TRAILING	[0] = (0,1,2); [1] = (2,1,3); [k even] = (k,k+1,k+2) [k odd] = (k+1,k,k+2) [N-3] = (see above)	"Odd" triangles have vertices reordered to yield increasing trailing vertices starting with v2.
TRISTRIP_REV (N >= 3) REORDER_LEADING	[0] = (0,2,1) [1] = (1,2,3);...; [k even] = (k,k+2,k+1) [k odd] = (k,k+1,k+2) [N-3] = (see above)	"Even" triangles have vertices reordered to yield increasing leading vertices starting with v0.
TRISTRIP_REV (N >= 3) REORDER_TRAILING	[0] = (1,0,2) [1] = (1,2,3);...; [k even] = (k+1,k,k+2) [k odd] = (k,k+1,k+2) [N-3] = (see above)	"Even" triangles have vertices reordered to yield increasing trailing vertices starting with v2.
TRISTRIP_ADJ (N even, N >= 6) REORDER_LEADING	N = 6 or 7: [0] = (0,2,4) N = 8 or 9: [0] = (0,2,4); [1] = (2,6,4); ...; N > 10: [0] = (0,2,4); [1] = (2,6,4); ...; [k>1, even] = (2k, 2k+2, 2k+4); [k>2, odd] = (2k, 2k+4, 2k+2);...; Trailing object:	"Odd" objects have vertices reordered to yield increasing-by-2 leading vertices starting with v0.

PrimTopologyType <PRIMITIVE_TOPOLOGY> (N = # of vertices)	Order of Vertices Streamed Out [<object#>] = (<vert#>,...);	Any SOL Notes
	[(N/2)-3, even] = (N-6,N-4,N-2); [(N/2)-3, odd] = (N-6,N-2,N-4);	
TRISTRIP_ADJ (N even, N >= 6) REORDER_TRAILING	N = 6 or 7: [0] = (0,2,4) N = 8 or 9: [0] = (0,2,4); [1] = (4,2,6); ...; N > 10: [0] = (0,2,4); [1] = (4,2,6); ...; [k>1, even] = (2k, 2k+2, 2k+4); [k>2, odd] = (2k+2,2k, 2k+4,);...; Trailing object: [(N/2)-3, even] = (N-6,N-4,N-2); [(N/2)-3, odd] = (N-4,N-6,N-2);	"Odd" objects have vertices reordered to yield increasing-by-2 trailing vertices starting with v4.
TRIFAN (N > 2)	[0] = (0,1,2); [1] = (0,2,3); ...; [N-3] = (0, N-2, N-1);	
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON, POLYGON_CONT	Same as TRIFAN	POLYGON_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
QUADLIST QUADSTRIP	N/A	Not supported after VF.
PATCHLIST_1	[0] = (0); [1] = (1); ...; [N-2] = (N-2);	
PATCHLIST_2	[0] = (0,1); [1] = (2,3); ...; [(N/2)-1] = (N-2,N-1)	
PATCHLIST_3..32	similar to above	

Stream Output Function

As previously mentioned, incoming 3D topologies are targeted at one of the four streams. The SOL stage contains state information specific to each of the four streams.

A stream's list of SO declarations (SO_DECL structures) is used to perform the SO function for objects targeted to that particular stream. The 3DSTATE_SO_DECL_LIST command is used to specify the list of SO_DECL structures for all four streams in parallel. Software is required to scan the SO_DECL lists of streams to determine which SO buffers are targeted. The Stream To Buffer Selects bits in 3DSTATE_SO_DECL_LIST must be programmed accordingly (if the buffer is targeted, the select bit must set, else it must be cleared).

If a stream has no SO_DECL state defined (NumEntries is 0), incoming objects targeting that stream are effectively ignored. As there is no attempt to perform stream output, overflow detection is neither required nor performed.

Otherwise, an overflow check is performed. First any attempt to output to a disabled buffer is detected. This occurs when the stream has a Stream To Buffer Selects bit set but the corresponding SO Buffer Enable is clear. Assuming all targeted buffers are enabled, an additional check is made to ensure that there is enough room in each targeted buffer to hold the number of vertices which be output to it (for the input object). Here the buffer's current end address is compared to what the write offset would be if the output was performed. The latter value is computed as (write_offset + vertex_count * buffer_pitch). If this value is greater than the end address, an overflow is signaled. This check is performed for each buffer included in Stream To Buffer Selects.

If an overflow is not signaled, the SO function is performed. The SO_DECL list for the targeted stream is traversed independently for each object vertex, and the operation specified by the SO_DECL structure is performed (typically causing data to be appended to an SO buffer). In the process, SO buffer Write Offsets are incremented.

Stream Output Buffers

Up to four SO buffers are supported. The SO buffer parameters (start/end address, etc.) are specified by the 3DSTATE_SO_BUFFER_INDEX_# commands.

The 3DSTATE_STREAMOUT command specifies an SO Buffer Enable bit for each of the buffers. If a buffer is disabled, its state is ignored, and no output will be attempted for that buffer. Any attempt to output to that buffer will immediately signal an overflow condition.

The SOL stage maintains a current Write Offset register value for each SO buffer. These registers can be written via MI_LOAD_REGISTER_MEM or MI_LOAD_REGISTER_IMM commands. The SOL stage will increment the Write Offsets as a part of the SO function. Software can cause a Write Offset register to be written to memory via an MI_STORE_REGISTER_MEM command, though a preceding flush operation may be required to ensure that any previous SO functions have completed.

Surface Format Name
R32G32B32A32_FLOAT
R32G32B32A32_SINT

Surface Format Name
R32G32B32A32_UINT
R32G32B32_FLOAT
R32G32B32_SINT
R32G32B32_UINT
R32G32_FLOAT
R32G32_SINT
R32G32_UINT
R32_SINT
R32_UINT
R32_FLOAT

Rendering Disable

Independent of SOL function enable, if rendering (i.e, 3D pipeline functions past the SOL stage) is enabled (via clearing the Rendering Disable bit), the SOL stage will pass topologies for a specific input stream (as selected by Render Stream Select) down the pipeline, with the exception of PATCHLIST_n topologies which are never passed downstream. Software must ensure that the vertices exiting the SOL stage include a vertex header and position value so that the topologies can be correctly processed by subsequent pipeline stages. Specifically, rendering must be disabled whenever 128-bit vertices are output from a GS thread.

If Rendering Disable is set, the SOL stage will prevent any topologies from exiting the SOL stage.

Statistics

The SOL stage controls the incrementing of two 64-bit statistics counter registers for each of the four output buffer slots, SO_NUM_PRIMS_WRITTEN[] and SO_PRIM_STORAGE_NEEDED.

3D Pipeline Rasterization

Common Rasterization State

This section contains rasterization state pointers.

Pointers
3DSTATE_VIEWPORT_STATE_POINTERS_CC
3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP
3DSTATE_SCISSOR_STATE_POINTERS
3DSTATE_RASTER

3D Pipeline - CLIP Stage Overview

The CLIP stage of the 3D Pipeline is similar to the GS stage in that it can be used to perform general processing on incoming 3D objects via spawned threads. However, the CLIP stage also includes

specialized logic to perform a *ClipTest* function on incoming objects. These two usage models of the CLIP stage are outlined below.

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Overview* chapter for a general description of a 3D Pipeline stage, as much of the CLIP stage operation and control falls under these "common" functions. I.e., many of the CLIP stage state variables and CLIP thread payload parameters are described in *3D Overview*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the CLIP stage, details on the ClipTest function, and any exceptions the CLIP stage exhibits with respect to common FF unit functions.

Clip Stage - 3D Clipping

The ClipTest fixed function is provided to optimize the CLIP stage for support of generalized *3D Clipping*. The CLIP FF unit examines the position of incoming vertices, performs a fixed function *VertexClipTest* on these positions, and then examines the results for the vertices of each independent object in *ClipDetermination*.

The results of ClipDetermination indicate whether an object is to be clipped (MustClip), discarded (TrivialReject) or passed down the pipeline unmodified (TrivialAccept). In the MustClip case, the fixed function clipping hardware is responsible for performing the actual 3D Clipping algorithm. The CLIP hardware is passed the source object vertex data and is able to output a new, arbitrary 3D primitive (e.g., the clipped primitive), or no output at all. Note that the output primitive is comprised of newly-generated vertex positions, barycentric attributes and shares vertices with the source primitive for rest of the attributes. The CLIP unit maintains the proper ordering of CLIP-generated primitives and any surrounding trivially-accepted primitives and processes all the primitives in order.

The outgoing primitive stream is sent down the pipeline to the Strip/Fan (SF) FF stage (now including the read back VUE Vertex Header data such as Vertex Position (NDC or screen space), RTAIndex, VPIndex, PointWidth) and control information (PrimType, PrimStart, PrimEnd) while the remainder of the vertex data remains in the VUE in the URB.

Fixed Function Clipper

The GPU supports Fixed Function Clipping.

Note: In an earlier generation, clipping was done in the EU. However, the clipper thread latency was high and caused a bottleneck in the pipeline. Hence the motivation for a fixed function clipper.

Concepts

This section provides an overview of 3D clip-testing and clipping concepts, as defined by the Direct3D* and OpenGL* APIs. It is provided as background material. Some of the concepts impact HW functionality while others impact CLIP kernel functionality.

* Other names and brands may be claimed as the property of others.

CLIP Stage Input

As a stage of the 3D pipeline, the CLIP stage receives inputs from the previous (GS) stage. Refer to *3D Overview* for an overview of the various types of input to a 3D Pipeline stage. The remainder of this subsection describes the inputs specific to the CLIP stage.

State

This section contains state clips for the Clip Stage. For each processor generation, the state used by the clip stage is defined by the appropriate inline state packet, linked below.

3DSTATE_CLIP

3D_STATE_CLIP

The Clip unit will unconditionally reject incoming PATCHLIST topologies, if not already discarded by SOL. So there is no need for SW to explicitly set the CLIP_mode to reject PATCHLIST topologies.

Clip Unit also receives 3DSTATE_RASTER. It also receives 3DSTATE_INT which is transparent to SW. 3DSTATE_INT provides 3DSTATE_VS, 3DSTATE_DS and 3DSTATE_GS fields.

Signal	Description	Formula
CLIP_INT::Front_Winding	Determines whether a triangle object is considered "front facing" if the screen space vertex positions, when traversed in the order, result in a clockwise (CW) or counter-clockwise (CCW) winding order. Does not apply to points or lines.	= 3DSTATE_RASTER::FrontWinding
CLIP_INT::CullMode	Controls removal (culling) of triangle objects based on orientation. The cull mode only applies to triangle objects and does not apply to lines, points, or rectangles.	= 3DSTATE_RASTER::CullMode
CLIP_INT::Viewport Z ClipTest Enable	This field is used to control whether the Viewport Z extents (near, far) are considered in VertexClipTest.	= 3DSTATE_RASTER::Viewport Z ClipTest Enable
CLIP_INT::User Clip Distance Cull Test Enable Bitmask	This 8-bit mask field selects which of the 8 user clip distances against which trivial reject/trivial accept determination needs to be made (does not cause a must clip). DX10 allows simultaneous use of ClipDistance and Cull Distance test of up to 8 distances. This mask must be mutually exclusive to final CLIP_INT::User Clip Distance Clip Test Enable Bitmask. Same mask bit can't be set for both.	= (3DSTATE_CLIP::ForceUser Clip Distance Cull Test Enable Bitmask == Force) ? 3DSTATE_CLIP::User Clip Distance Cull Test Enable Bitmask : 3DSTATE_GS::GS_Enable ? 3DSTATE_GS:: GS User Clip Distance Cull Test Enable Bitmask : 3DSTATE_DS:DS_Enable ? 3DSTATE_DS:: DS User Clip Distance Cull Test Enable Bitmask : 3DSTATE_INT:VS_Enable ?

Signal	Description	Formula
		3DSTATE_VS:: VS User Clip Distance Cull Test Enable Bitmask : 0
CLIP_INT::User Clip Distance Clip Test Enable Bitmask	This 8-bit mask field selects which of the 8 user clip distances against which trivial reject/trivial accept determination needs to be made (does not cause a must clip). DX10 allows simultaneous use of ClipDistance and Clip Distance test of up to 8 distances. This mask must be mutually exclusive to final CLIP_INT::User Clip Distance Cull Test Enable Bitmask. Same mask bit can't be set for both.	= (3DSTATE_CLIP::ForceUser Clip Distance Clip Test Enable Bitmask == Force) ? 3DSTATE_CLIP::User Clip Distance Clip Test Enable Bitmask : 3DSTATE_GS:GS_Enable ? 3DSTATE_GS::GS User Clip Distance Clip Test Enable Bitmask : 3DSTATE_DS::DS_Enable ? 3DSTATE_DS::DS User Clip Distance Clip Test Enable Bitmask : 3DSTATE_VS::VS_Enable ? 3DSTATE_VS::VS User Clip Distance Clip Test Enable Bitmask : 0

VUE Readback

Starting with the CLIP stage, the 3D pipeline requires vertex information in addition to the VUE handle. For example, the CLIP unit's VertexClipTest function needs the vertex position, as does the SF unit's functions. This information is obtained by the 3D pipeline reading a portion of each vertex's VUE data directly from the URB. This readback (effectively) occurs immediately before the CLIP VertexClipTest function, and immediately after a CLIP thread completes the output of a destination VUE.

The Vertex Header (first 256 bits) of the VUE data is read back. (See the previous *VUE Formats* subsection (above) for details on the content and format of the Vertex Header.) Additional Clip/Cull data (located immediately past the Vertex Header) may be read prior to clipping.

This readback occurs automatically and is not under software control. The only software implication is that the Vertex Header must be valid at the readback points, and therefore must have been previously loaded or written by a thread.

VertexClipTest Function

The VertexClipTest function compares each incoming vertex position (x,y,z,w) with various viewport and guardband parameters (either hard-coded values or specified by state variables).

The RHW component of the incoming vertex position is tested for NaN value, and if a NaN is detected, the vertex is marked as "BAD" by setting the outcode[BAD]. If a NaN is detected in any vertex

homogeneous x,y,z,w component or an enabled ClipDistance value, the vertex is marked as "BAD" by setting the outcode[BAD].

In general, any object containing a BAD vertex will be discarded, as how to clip/render such objects is undefined.

However, in the case of CLIP_ALL mode, a CLIP thread will be spawned even for objects with "BAD" vertices. The CLIP kernel is required to handle this case and can examine the **Object Outcode [BAD]** payload bit to detect the condition. (Note that the VP and GB Object Outcodes are UNDEFINED when BAD is set.)

If the incoming RHW coordinate is negative (including negative 0) the NEGW outcode is set. Also, this condition is used to select the proper comparison functions for the VP and GB outcode tests (below).

Next, the VPXY and GB outcodes are computed, depending on the corresponding enable SV bits. If one of VPXY or GB is disabled, the enabled set of outcodes are copied to the disabled set of outcodes. This effectively defines the disabled boundaries to coincide with the enabled boundaries (i.e., disabling the GB is just like setting it to the VPXY values, and vice versa).

The VPZ outcode is computed as required by the API mode SV.

Separate VP Z Near (ZMin) and Z Far (ZMax) ClipTest Enable state bits are provided in 3DSTATE_RASTER.

Finally, the incoming UserClipFlags are masked and copied to corresponding outcodes.

The following algorithm is used by VertexClipTest:

```
//
// Vertex ClipTest
//
// On input:
// if (CLIP.PreMapped)
//   x,y are viewport mapped
//   z is NDC ([0,1] is visible)
// else
//   x,y,z are NDC (post-perspective divide)
//   w is always 1/w
//
// Initialize outCodes to "inside"
//
outCode[*] = 0
//
// Check if w is NaN
// Any object containing one of these "bad" vertices will likely be discarded
//
if (ISNAN(homogeneous x,y,z,w or enabled ClipDistance value)
{
   outCode[BAD] = 1
}
//
// If 1/w is negative, w is negative and therefore outside of the w=0 plane
//
//
rhw_neg = ISNEG(rhw)
if (rhw_neg)
{
   outCode[NEGW] = 1
}
//
```

```
// View Volume Clip Test
// If Premapped, the 2D viewport is defined in screen space
// otherwise the canonical NDC viewvolume applies ([-1,1])
//
if (CLIP_STATE.PreMapped)
{
   vp_XMIN = CLIP_STATE.VP_XMIN
   vp_XMAX = CLIP_STATE.VP_XMAX
   vp_YMIN = CLIP_STATE.VP_YMIN
   vp_YMAX = CLIP_STATE.VP_YMAX
} else {
   vp_XMIN = -1.0f
   vp_XMAX = +1.0f
   vp_YMIN = -1.0f
   vp_YMAX = +1.0f
}

if (CLIP_STATE.ViewportXYClipTestEnable) {
   outCode[VP_XMIN] = (x < vp_XMIN)
   outCode[VP_XMAX] = (x > vp_XMAX)
   outCode[VP_YMIN] = (y < vp_YMIN)
   outCode[VP_YMAX] = (y > vp_YMAX)

   #ifdef (BW-E0)
   if (rhw_neg) {
      outCode[VP_XMIN] = (x >= vp_XMIN)
      outCode[VP_XMAX] = (x <= vp_XMAX)
      outCode[VP_YMIN] = (y >= vp_XMIN)
      outCode[VP_YMAX] = (y <= vp_XMAX)
   }
   #endif
   if (rhw_neg) {
      outCode[VP_XMIN] = (x > vp_XMIN)
      outCode[VP_XMAX] = (x < vp_XMAX)
      outCode[VP_YMIN] = (y > vp_XMIN)
      outCode[VP_YMAX] = (y < vp_XMAX)
   }
}

if (CLIP_STATE.ViewportZClipTestEnable) {
   if (CLIP_STATE.APIMode == APIMODE_D3D) {
      vp_ZMIN = 0.0f
      vp_ZMAX = 1.0f
   } else { // OGL
      vp_ZMIN = -1.0f
      vp_ZMAX = 1.0f
   }
   outCode[VP_ZMIN] = (z < vp_ZMIN)
   outCode[VP_ZMAX] = (z > vp_ZMAX)

   #ifdef (BW-E0)
   if (rhw_neg) {
      outCode[VP_ZMIN] = (z >= vp_ZMIN)
      outCode[VP_ZMAX] = (z <= vp_ZMAX)
   }
   #endif
   if (rhw_neg) {
      outCode[VP_ZMIN] = (z > vp_ZMIN)
      outCode[VP_ZMAX] = (z < vp_ZMAX)
   }
}
//
// Guardband Clip Test
//
```

```
if {CLIP_STATE.GuardbandClipTestEnable) {
   gb_XMIN = CLIP_STATE.Viewport[vpindex].GB_XMIN
   gb_XMAX = CLIP_STATE.Viewport[vpindex].GB_XMAX
   gb_YMIN = CLIP_STATE.Viewport[vpindex].GB_YMIN
   gb_YMAX = CLIP_STATE.Viewport[vpindex].GB_YMAX
   outCode[GB_XMIN] = (x < gb_XMIN)
   outCode[GB_XMAX] = (x > gb_XMAX)
   outCode[GB_YMIN] = (y < gb_YMIN)
   outCode[GB_YMAX] = (y > gb_YMAX)

   #ifdef (BW-E0)
   if (rhw_neg) {
      outCode[GB_XMIN] = (x >= gb_XMIN)
      outCode[GB_XMAX] = (x <= gb_XMAX)
      outCode[GB_YMIN] = (y >= gb_YMIN)
      outCode[GB_YMAX] = (y <= gb_YMAX)
   }
   #endif
   if (rhw_neg) {
      outCode[GB_XMIN] = (x > gb_XMIN)
      outCode[GB_XMAX] = (x < gb_XMAX)
      outCode[GB_YMIN] = (y > gb_YMIN)
      outCode[GB_YMAX] = (y < gb_YMAX)
   }
}
//
// Handle case where either VP or GB disabled (but not both)
// In this case, the disabled set take on the outcodes of the enabled set
//
if (CLIP_STATE.ViewportXYClipTestEnable && !CLIP_STATE.GuardbandClipTestEnable) {
   outCode[GB_XMIN] = outCode[VP_XMIN]
   outCode[GB_XMAX] = outCode[VP_XMAX]
   outCode[GB_YMIN] = outCode[VP_YMIN]
   outCode[GB_YMAX] = outCode[VP_YMAX]
} else if (!CLIP_STATE.ViewportXYClipTestEnable && CLIP_STATE.GuardbandClipTestEnable) {
   outCode[VP_XMIN] = outCode[GB_XMIN]
   outCode[VP_XMAX] = outCode[GB_XMAX]
   outCode[VP_YMIN] = outCode[GB_YMIN]
   outCode[VP_YMAX] = outCode[GB_YMAX]
}
//
// X/Y/Z NaN Handling
//
xyorgben = (CLIP_STATE.ViewportXYClipTestEnable || CLIP_STATE.GuardbandClipTestEnable)
if (isNAN(x)) {
   outCode[GB_XMIN] = xyorgben
   outCode[GB_XMAX] = xyorgben
   outCode[VP_XMIN] = xyorgben
   outCode[VP_XMAX] = xyorgben
}
if (isNAN(y)) {
   outCode[GB_YMIN] = xyorgben
   outCode[GB_YMAX] = xyorgben
   outCode[VP_YMIN] = xyorgben
   outCode[VP_YMAX] = xyorgben
}
if (isNaN) {
   outCode[VP_ZMIN] = CLIP_STATE.ViewportZClipTestEnable
   outCode[VP_ZMAX] = CLIP_STATE.ViewportZClipTestEnable
}
//
// UserClipFlags
//
ExamineUCFs
```

```
for (i=0; i<7; i++)
{
   outCode[UC0+i] = userClipFlag[i] & CLIP_STATE.UserClipFlagsClipTestEnableBitmask[i]
}
outCode[UC7] = userClipFlag[i] & CLIP_STATE.UserClipFlagsClipTestEnableBitmask[7]
```

Object Staging

The CLIP unit's Object Staging Buffer (OSB) accepts streams of input vertex information packets, along with each vertex's VertexClipTest result (outCode). This information is buffered until a complete object, or the last vertex of the primitive topology is received. The OSB then performs the ClipDetermination function on the object vertices, and takes the actions required by the results of that function.

Partial Object Removal

The OSB is responsible for removing incomplete LINESTRIP and TRISTRIP objects that it may receive from the preceding stage (GS). Partial object removal is not supported for other primitive types due to either (a) the GS is not permitted to output those primitive types (e.g., primitives with adjacency info), and the VF unit will have removed the partial objects as part of 3DPRIMITIVE processing, or (b) although the GS thread is allowed to output the primitive type (e.g., LINELIST), it is assumed that the GS kernel will be correctly implemented to avoid outputting partial objects (or pipeline behavior is UNDEFINED).

An object is considered 'partial' if the last vertex of the primitive topology is encountered (i.e., PrimEnd is set) before a complete set of vertices for that object have been received. Given that only LINESTRIP and TRISTRIP primitive types are subject to CLIP unit partial object removal, the only supported cases of partial objects are 1-vertex LINESTRIPs and 1 or 2-vertex TRISTRIPs.

ClipDetermination Function

In ClipDetermination, the vertex outcodes of the primitive are combined in order to determine the clip status of the object (TR: trivially reject; TA: trivial accept; MC: must clip; BAD: invalid coordinate). Only those vertices included in the object are examined (3 vertices for a triangle, 2 for a line, and 1 for a point). The outcode bit arrays for the vertices are separately ANDed (intersection) and ORed (union) together (across vertices, not within the array) to yield objANDCode and objORCode bit arrays.

TR/TA against interesting boundary subsets are then computed. The TR status is computed as the logical OR of the appropriate objANDCode bits, as the vertices need only be outside of one common boundary to be trivially rejected. The TA status is computed as the logical NOR of the appropriate objORCode bits, as any vertex being outside of any of the boundaries prevents the object from being trivially accepted.

If any vertex contains a BAD coordinate, the object is considered BAD and any computed TR/TA results will effectively be ignored in the final action determination.

Next, the boundary subset TR/TA results are combined to determine an overall status of the object. If the object is TR against any viewport or enabled UC plane, the object is considered TR. Note that, by definition, being TR against a VPXY boundary implies that the vertices will be TR against the corresponding GB boundary, so computing TR_GB is unnecessary.

The treatment of the UCF outcodes is conditional on the UserClipFlags MustClip Enable state. If DISABLED, an object that is not TR against the UCFs is considered TA against them. Put another way,

objects will only be culled (not clipped) with respect to the UCFs. If ENABLED, the UCF outcodes are treated like the other outcodes, in that they are used to determine TR, TA or MC status, and an object can be passed to a CLIP thread simply based on it straddling a UCF.

Finally, the object is considered MC if it is neither TR or TA.

The following logic is used to compute the final TR, TA, and MC status.

```
//
// ClipDetermination
//
// Compute objANDCode and objORCode
//
switch (object type) {
   case POINT:
      {
         objANDCode[...] = v0.outCode[...]
         objORCode[...]  = v0.outCode[...]
      } break
   case LINE:
      {
         objANDCode[...] = v0.outCode[...] & v1.outCode[...]
         objORCode[...]  = v0.outCode[...] | v1.outCode[...]
      } break
   case TRIANGLE:
      {
         objANDCode[...] = v0.outCode[...] & v1.outCode[...] & v2.outCode[...]
         objORCode[...]  = v0.outCode[...] | v1.outCode[...] | v2.outCode[...]
      } break
}
//
// Determine TR/TA against interesting boundary subsets
//
TR_VPXY = (objANDCode[VP_L] | objANDCode[VP_R] | objANDCode[VP_T] | objANDCode[VP_B])
TR_GB   = (objANDCode[GB_L] | objANDCode[GB_R] | objANDCode[GB_T] | objANDCode[GB_B])
TA_GB   = !(objORCode[GB_L] | objORCode[GB_R]  | objORCode[GB_T]  | objORCode[GB_B])
TA_VPZ  = !(objORCode[VP_N] | objORCode[VP_Z])
TR_VPZ  = (objANDCode[VP_N] | objANDCode[VP_Z])
TA_UC   = !(objORCode[UC0]  | objORCode[UC1]   | ... | objORCode[UC7])
TR_UC   = (objANDCode[UC0]  | objANDCode[UC1]  | ... | objANDCode[UC7])
BAD     = objORCode[BAD]
TA_NEGW = !objORCode[NEGW]
TR_NEGW = objANDCode[NEGW]

//
// Trivial Reject
//
// An object is considered TR if all vertices are TR against any common boundary
// Note that this allows the case of the VPXY being outside the GB
//
TR = TR_GB || TR_VPXY || TR_VPZ || TR_UC || TR_NEGW
#else
TR = TR_GB || TR_VPXY || TR_VPZ || TR_UC

//
// Trivial Accept
//
// For an object to be TA, it must be TA against the VPZ and GB, not TR,
// and considered TA against the UC planes and NEGW
// If the UCMC mode is disabled, an object is considered TA against the UC
// as long as it isn't TR against the UC.
// If the UCMC mode is enabled, then the object really has to be TA against the UC
// to be considered TA
```

```
// In this way, enabling the UCMC mode will force clipping if the object is neither
// TA or TR against the UC
//
TA   = !TR && TA_GB && TA_VPZ && TA_NEGW
UCMC = CLIP_STATE.UserClipFlagsMustClipEnable
TA   = TA && ( (UCMC && TA_UC) || (!UCMC && !TR_UC) )

//
// MustClip
// This is simply defined as not TA or TR
// Note that exactly one of TA, TR and MC will be set
//
MC = !(TA || TR)
```

ClipMode State

The ClipMode state determines what action the CLIP unit takes given the results of ClipDetermination. The possible actions are:

- **PASSTHRU:** Pass the object directly down the pipeline. A CLIP thread is not spawned.
- **DISCARD:** Remove the object from the pipeline and dereference object vertices as required (that is, dereferencing will not occur if the vertices are shared with other objects).
- **SPAWN:** Pass the object to a CLIP thread. In the process of initiating the thread, the object vertices may be dereferenced.

The following logic is used to determine what to do with the object (PASSTHRU or DISCARD or SPAWN).

```
//
// Use the ClipMode to determine the action to take
//
switch (CLIP_STATE.ClipMode) {
   case NORMAL:
      {
         PASSTHRU = TA && !BAD
         DISCARD = TR || BAD
         SPAWN   = MC && !BAD
      }
   case CLIP_ALL:
      {
         PASSTHRU = 0
         DISCARD  = 0
         SPAWN    = 1
      }
   case CLIP_NOT_REJECT:
      {
         PASSTHRU = 0
         DISCARD  = TR   || BAD
         SPAWN    = !(TR || BAD)
      }
   case REJECT_ALL:
      {
         PASSTHRU = 0
         DISCARD  = 1
         SPAWN    = 0
      }
   case ACCEPT_ALL:
      {
         PASSTHRU = !BAD
         DISCARD  = BAD
         SPAWN    = 0
```

```
        }
} endswitch
```

NORMAL ClipMode

In NORMAL mode, objects will be discarded if TR or BAD, passed through if TA, and passed to a CLIP thread if MC. Those mode is typically used when the CLIP kernel is only used to perform 3D Clipping (the expected usage model).

CLIP_ALL ClipMode

In CLIP_ALL mode, all objects (regardless of classification) will be passed to CLIP threads. Note that this includes BAD objects. This mode can be used to perform arbitrary processing in the CLIP thread, or as a backup if for some reason the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are not sufficient for controlling 3D Clipping.

CLIP_NON_REJECT ClipMode

This mode is similar to CLIP_ALL mode, but TR and BAD objects are discarded an all other (TA, MC) objects are passed to CLIP threads. Usage of this mode assumes that the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are sufficient at least in respect to determining trivial reject.

REJECT_ALL ClipMode

In REJECT_ALL mode, all objects (regardless of classification) are discarded. This mode effectively clips out all objects.

ACCEPT_ALL ClipMode

In ACCEPT_ALL mode, all non-BAD objects are passed directly down the pipeline. This mode partially disables the CLIP stage. BAD objects will still be discarded, and incomplete primitives (generated by a GS thread) will be discarded.

Primitive topologies with adjacency are also handled, in that the adjacent-only vertices are dereferenced, and only non-adjacent objects are passed down the pipeline. This condition can arise when primitive topologies with adjacency are generated but the GS stage is disabled. If this condition is allowed, the CLIP stage must not be completely disabled - as this would allow adjacent vertices to pass through the CLIP stage and lead to unpredictable results as the rest of the pipeline does not comprehend adjacency.

Object Pass-Through

Depending on ClipMode, objects may be passed directly down the pipeline. The PrimTopologyType associated with the output objects may differ from the input PrimTopologyType, as shown in the table below.

Programming Note: The CLIP unit does *not* tolerate primitives with adjacency that have "dangling vertices". This should not be an issue under normal conditions, as the VF unit does not generate these sorts of primitives and the GS thread is restricted (though by specification only) to not output these sorts of primitives.

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
POINTLIST	POINTLIST	
POINTLIST_BF	POINTLIST_BF	
LINELIST	LINELIST	
LINELIST_ADJ	LINELIST	Adjacent vertices removed.
LINESTRIP	LINESTRIP	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT	LINESTRIP	Adjacent vertices removed. LINESTRIP_ADJ_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
LINESTRIP_BF	LINESTRIP_BF	
LINESTRIP_CONT	LINESTRIP_CONT	
LINESTRIP_CONT_BF	LINESTRIP_CONT_BF	
LINELOOP	N/A	Not supported after GS.
TRILIST	TRILIST	
RECTLIST	RECTLIST	
TRILIST_ADJ	TRILIST	Adjacent vertices removed.
TRISTRIP	TRISTRIP or TRISTRIP_REV	Depends on where the incoming strip is broken (if at all) by discarded or clipped objects See Tristrip Clipping subsection.
TRISTRIP_REV	TRISTRIP or TRISTRIP_REV	Depends on where the incoming strip is broken (if at all) by discarded or clipped objects. See Tristrip Clipping subsection.
TRISTRIP_ADJ	TRISTRIP or TRISTRIP_REV	Depends on where the incoming strip is broken (if at all) by discarded or clipped objects. Adjacent vertices removed. See Tristrip Clipping subsection.
TRIFAN	TRIFAN	
TRIFAN_NOSTIPPLE	TRIFAN_NOSTIPPLE	
POLYGON, POLYGON_CONT	POLYGON	POLYGON_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
QUADLIST	N/A	Not supported after GS.
QUADSTRIP	N/A	Not supported after GS.

Primitive Output

(This section refers to output from the CLIP unit to the pipeline, not output from the CLIP thread)

The CLIP unit will output primitives (either passed-through or generated by a CLIP thread) in the proper order. This includes the buffering of a concurrent CLIP thread's output until the preceding CLIP thread terminates. Note that the requirement to buffer subsequent CLIP thread output until the preceding CLIP thread terminates has ramifications on determining the number of VUEs allocated to the CLIP unit and the number of concurrent CLIP threads allowed.

Other Functionality

Statistics Gathering

Software is responsible for controlling (enabling) these counters in order to provide the required statistics at the DDI level. For example, software might need to disable statistics gathering before submitting non-API-visible objects (e.g., RECTLISTs) for processing.

The CLIP unit must be ENABLED (via the CLIP Enable bit of PIPELINED_STATE_POINTERS) for it to affect the statistics counters. This might lead to a pathological case where the CLIP unit needs to be ENABLED simply to provide statistics gathering. If no clipping functionality is desired, Clip Mode can be set to ACCEPT_ALL to effectively inhibit clipping while leaving the CLIP stage ENABLED.

The statistic the CLIP unit affects (if enabled) is CL_INVOCATION_COUNT, incremented for every object received from the GS stage.

CL_INVOCATION_COUNT

If the **Statistics Enable** bit (CLIP_STATE) is set, the CLIP unit increments the CL_INVOCATION_COUNT register for every complete object received from the GS stage.

To maintain a count of application-generated objects, software must clear the CLIP unit's **Statistic Enable** whenever driver-generated objects are rendered.

3D Pipeline - Strips and Fans (SF) Stage

The Strips and Fan (SF) stage of the 3D pipeline is responsible for performing "setup" operations required to rasterize 3D objects.

This functionality is handled completely in hardware, and the SF unit no longer has the ability to spawn threads.

Inputs from CLIP

The following table describes the per-vertex inputs passed to the SF unit from the previous (CLIP) stage of the pipeline.

SF's Vertex Pipeline Inputs

Variable	Type	Description
primType	enum	Type of primitive topology the vertex belongs to. SF-Supported Primitive Types & Vertex Count Restrictions for a list of primitive types supported by the SF unit. See *3D Pipeline* for descriptions of these topologies. Notes: The CLIP unit will convert any primitive with adjacency (3DPRIMxxx_ADJ) it receives from the pipeline into the corresponding primitive without adjacency (3DPRIMxxx). QUADLIST, QUADSTRIP, LINELOOP primitives are not supported by the SF unit. Software must use a GS thread to convert these to some other (supported) primitive type. If an object is clipped by the hardware clipper, the CLunit would force this field to "3DPRIM_POLYGON". SFunit would process this incoming object just as it would any other "3DPRIM_POLYGON". SFunit selects vertex 0 as the provoking vertex.
primStart,primEnd	boolean	Indicate vertex's position within the primitive topology
vInX[]	float	Vertex X position (screen space or NDC space)
vInY[]	float	Vertex Y position (screen space or NDC space)
vInZ[]	float	Vertex Z position (screen space or NDC space)
vInInvW[]	float	Reciprocal of Vertex homogeneous (clip space) W
hVUE[]	URB address	Points to the vertex's data stored in the URB (one VUE handle per vertex)
renderTargetArrayIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in subsequent processing.
viewPortIndex	uint	Index of a viewport transform matrix within the SF_VIEWPORT structure used to perform Viewport Transformation on object vertices and scissor operations on an object. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in the Viewport Transform and Scissor subfunctions, otherwise the value is ignored. Note that for primitive topologies with vertices shared between objects, this means a shared vertex may be subject to multiple Viewport Transformation operations if the viewPortIndex varies

Variable	Type	Description
		within the topology.
pointSize	uint	If this vertex is within a POINTLIST[_BF] primitive topology, this value specifies the screen space size (width,height) of the square point to be rasterized about the vertex position. Otherwise, the value is ignored.

Attribute Setup/Interpolation Process

The following sections describe the Attribute Setup/Interpolation Process.

Attribute Setup/Interpolation Process

Hardware computes all needed parameters, as there is no setup thread.

Outputs to WM

The outputs from the SF stage to the WM stage are mostly comprised of implementation-specific information required for the rasterization of objects. The types of information is summarized below, but as the interface is not exposed to software a detailed discussion is not relevant to this specification.

- PrimType of the object
- VPIndex, RTAIndex associated with the object
- Coefficients for Z, 1/W, perspective and non-perspective b1 and b2 per vertex, and attribute vertex deltas a0, a1, and a2 per attribute.
- Information regarding the X,Y extent of the object (e.g., bounding box, etc.).
- Edge or line interpolation information (e.g., edge equation coefficients, etc.).
- Information on where the WM is to start rasterization of the object.
- Object orientation (front/back-facing).
- Last Pixel indication (for line drawing).

Primitive Assembly

The first subfunction within the SF unit is *Primitive Assembly*. Here 3D primitive vertex information is buffered and, when a sufficient number of vertices are received, converted into basic 3D objects which are then passed to the Viewport Transformation subfunction.

The number of vertices passed with each primitive is constrained by the primitive type. Passing any other number of vertices results in UNDEFINED behavior. Note that this restriction only applies to primitive output by GS threads (which is under control of the GS kernel). See the Vertex Fetch chapter for details on how the VF unit automatically removes incomplete objects resulting from processing a 3DPRIMITIVE command.

SF-Supported Primitive Types & Vertex Count Restrictions

primType	VertexCount Restriction
3DPRIM_TRILIST	nonzero multiple of 3
3DPRIM_TRISTRIP 3DPRIM_TRISTRIP_REVERSE	>=3
3DPRIM_TRIFAN 3DPRIM_TRIFAN_NOSTIPPLE 3DPRIM_POLYGON	>=3
3DPRIM_LINELIST	nonzero multiple of 2
3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	>=2
3DPRIM_RECTLIST	nonzero multiple of 3
3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF	nonzero

3D Object Types for a list of the 3D object types.

3D Object Types

objectType	generated by primType	Vertices/Object
3DOBJ_POINT	3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF	1
3DOBJ_LINE	3DPRIM_LINELIST 3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	2
3DOBJ_TRIANGLE	3DPRIM_TRILIST 3DPRIM_TRISTRIP 3DPRIM_TRISTRIP_REVERSE 3DPRIM_TRIFAN 3DPRIM_TRIFAN_NOSTIPPLE 3DPRIM_POLYGON	3

objectType	generated by primType	Vertices/Object
3DOBJ_RECTANGLE	3DPRIM_RECTLIST	3 (expanded to 4 in RectangleCompletion)

Primitive Decomposition Outputs for the outputs of Primitive Decomposition.

Primitive Decomposition Outputs

Variable	Type	Description
objectType	enum	Type of object. 3D Object Types
nV	uint	The number of object vertices passed to Object Setup. 3D Object Types
v[0..nV-1]*	various	Data arrays associated with object vertices. Data in the array consists of X, Y, Z, invW and a pointer to the other vertex attributes. These additional attributes are not used by directly by the 3D fixed functions but are made available to the SF thread. The number of valid vertices depends on the object type. *Primitive Assembly*
invertOrientation	enum	Indicates whether the orientation (CW or CCW winding order) of the vertices of a triangle object should be inverted. Ignored for non-triangle objects.
backFacing	enum	Valid only for points and line objects, indicates a back facing object. This is used later for culling.
provokingVtx	uint	Specifies the index (into the *v[]* arrays) of the vertex considered the "provoking" vertex (for flat shading). The selection of the provoking vertex is programmable via SF_STATE (**xxx Provoking Vertex Select** state variables.)
polyStippleEnable	boolean	TRUE if Polygon Stippling is enabled. FALSE for TRIFAN_NOSTIPPLE. Ignored for non-triangle objects.
continueStipple	boolean	Only applies to line objects. TRUE if Line Stippling should be continued (i.e., not reset) from where the previous line left off. If FALSE, Line Stippling is reset for each line object.
renderTargetIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. This value is simply passed in SF thread payloads and not used within the SF unit.
viewPortIndex	uint	Index of a viewport transform matrix within the SF_VIEWPORT structure used to perform Viewport Transformation on object vertices and scissor operations on an object.
pointSize	unit	For point objects, this value specifies the screen space size (width,height) of the square point to be rasterized about the vertex position. Otherwise, the value is ignored.

The following table defines, for each primitive topology type, which vertex's VPIndex/RTAIndex applies to the objects within the topology.

VPIndex/RTAIndex Selection

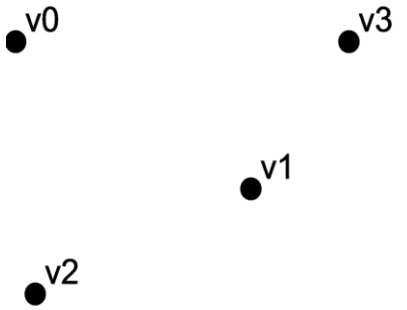
PrimTopologyType	Viewport Index Usage
POINTLIST POINTLIST_BF	Each vertex supplies the VPIndex for the corresponding point object
LINELIST	The leading vertex of each line supplies the VPIndex for the corresponding line object. V0.VPIndex-> Line(V0,V1) V2.VPIndex-> Line(V2,V3) ...
LINESTRIP LINESTRIP_BF LINESTRIP_CONT LINESTRIP_CONT_BF	The leading vertex of each line segment supplies the VPIndex for the corresponding line object. V0.VPIndex-> Line(V0,V1) V1.VPIndex-> Line(V1,V2) ... NOTE: If the VPIndex changes within the topology, shared vertices will be processed (mapped) multiple times.
TRILIST RECTLIST	The leading vertex of each triangle/rect supplies the VPIndex for the corresponding triangle/rect objects. V0.VPIndex-> Tri(V0,V1,V2) V3.VPIndex-> Tri(V3,V4,V5) ... NOTE: For Autostrips multiple viewport index is not supported. APIs defines the multiple viewport index to be output from Geometry shader that only generates Tristrips. If any other shader outputing multiple viewport indices for other topologies either autostrip needs to be disable or clipper guardband test needs to be disabled.
TRISTRIP TRISTRIP_REVERSE	The leading vertex of each triangle supplies the VPIndex for the corresponding triangle object. V0.VPIndex-> Tri(V0,V1,V2) V1.VPIndex-> Tri(V1,V2,V3) ... NOTE: If the VPIndex changes within the primitive, shared vertices will be processed (mapped) multiple times.

PrimTopologyType	Viewport Index Usage
TRIFAN TRIFAN_NOSTIPPLE POLYGON	The first vertex (V0) supplies the VPIndex for all triangle objects.

Point List Decomposition

The 3DPRIM_POINTLIST and 3DPRIM_POINTLIST_BACKFACING primitives specify a list of independent points.

3DPRIM_POINTLIST Primitive



The decomposition process divides the list into a series of basic 3DOBJ_POINT objects that are then passed individually and in order to the Object Setup subfunction. The *provokingVertex* of each object is, by definition, v[0].

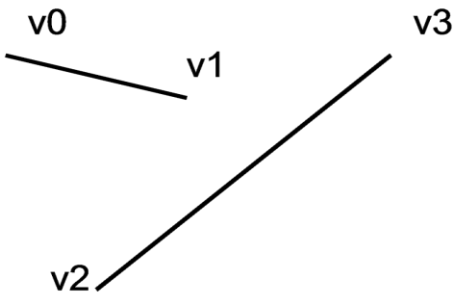
Points have no winding order, so the primitive command is used to explicitly state whether they are back-facing or front-facing points. Primitives of type 3DPRIM_POINTLIST_BACKFACING are decomposed exactly the same way as 3DPRIM_POINTLIST primitives, but the *backFacing* variable is set for resulting point objects being passed on to object setup.

```
PointListDecomposition()
{
   objectType = 3DOBJ_POINT
   nV = 1
   provokingVtx = 0
   if (primType == 3DPRIM_POINTLIST)
      {
         backFacing = FALSE
      }
   else // primType == 3DPRIM_POINTLIST_BACKFACING
      {
         backFacing = TRUE
      }

   for each (vertex in [0..vertexCount-1])
      {
         v[0] <- vIn[i] // copy all arrays
                        // (for example, v[]X, v[]Y, and so on)
         ObjectSetup()
      }
}
```

Line List Decomposition

The 3DPRIM_LINELIST primitive specifies a list of independent lines.

3DPRIM_LINELIST Primitive



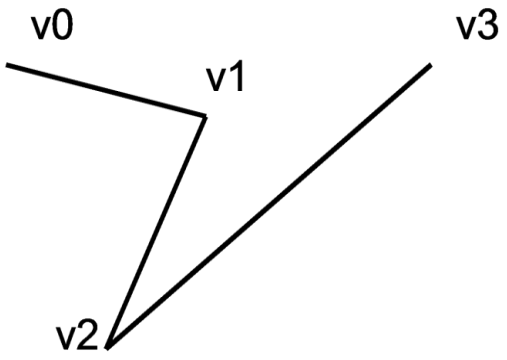
The decomposition process divides the list into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0, v1; v2, v3; and so on. The *provokingVertex* of each object is taken from the **Line List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

```
LineListDecomposition()
   {
      objectType = 3DOBJ_LINE
      nV = 2
      provokingVtx = Line List/Strip Provoking Vertex Select       continueStipple = FALSE
      for each (vertex I in [0..vertexCount-2] by 2)
         {
             v[0] arrays <- vIn[i] arrays
             v[1] arrays <- vIn[i+1] arrays
             ObjectSetup()
         }
   }
```

Line Strip Decomposition

The 3DPRIM_LINESTRIP, 3DPRIM_LINESTRIP_CONT, 3DPRIM_LINESTRIP_BF, and 3DPRIM_LINESTRIP_CONT_BF primitives specify a list of connected lines.

3DPRIM_LINESTRIP_xxx Primitive



The decomposition process divides the strip into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0,v1; v1,v2; and so on. The *provokingVertex* of each object is taken from the **Line List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

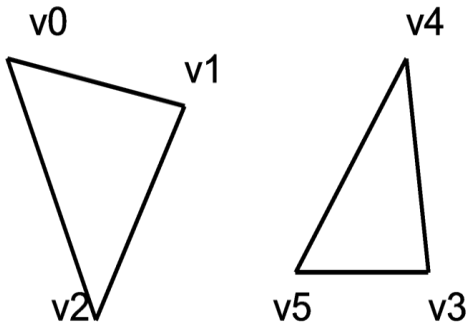
Lines have no winding order, so the primitive command is used to explicitly state whether they are back-facing or front-facing lines. Primitives of type 3DPRIM_LINESTRIP[_CONT]_BF are decomposed exactly the same way as 3DPRIM_LINESTRIP[_CONT] primitives, but the *backFacing* variable is set for the resulting line objects being passed on to object setup. Likewise 3DPRIM_LINESTRIP_CONT[_BF] primitives are decomposed identically to basic line strips, but the *continueStipple* variable is set to true so that the line stipple pattern will pick up from where it left off with the last line primitive, rather than being reset.

```
LineStripDecomposition()
   {
      objectType = 3DOBJ_LINE
      nV = 2
      provokingVtx = Line List/Strip Provoking Vertex Select
      if (primType == 3DPRIM_LINESTRIP)
         {
             backFacing = FALSE
             continueStipple = FALSE
         } else if (primType == 3DPRIM_LINESTRIP_BF)
            {
               backFacing = TRUE
               continueStipple = FALSE
            } else if (primType == 3DPRIM_LINESTRIP_CONT)
               {
                  backFacing = FALSE
                  continueStipple = TRUE
               } else if (primType == 3DPRIM_LINESTRIP_CONT_BF)
                  {
                     backFacing = TRUE
                     continueStipple = TRUE
                  }
      for each (vertex I in [0..vertexCount-1])
         {
             v[0] arrays <- vIn[i] arrays
             v[1] arrays <- vIn[i+1] arrays
             ObjectSetup()
             continueStipple = TRUE
         }
   }
```

Triangle List Decomposition

The 3DPRIM_TRILIST primitive specifies a list of independent triangles.

3DPRIM_TRILIST Primitive



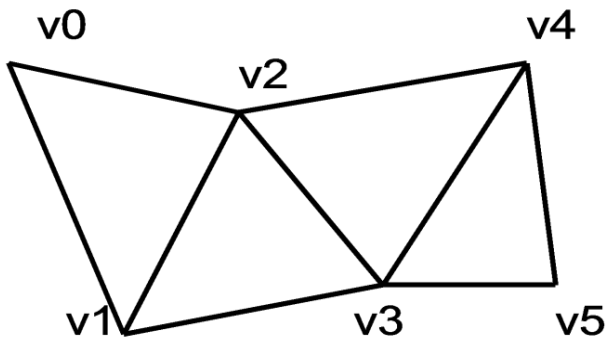
The decomposition process divides the list into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v3,v4,v5; and so on. The *provokingVertex* of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

```
TriangleListDecomposition() {
  objectType = 3DOBJ_TRIANGLE
      nV = 3
  invertOrientation = FALSE
  provokingVtx = Triangle List/Strip Provoking Vertex Select
  polyStippleEnable = TRUE
  for each (vertex I in [0..vertexCount-3] by 3)
      {
          v[0] arrays <- vIn[i] arrays
          v[1] arrays <- vIn[i+1] arrays
          v[2] arrays <- vIn[i+2] arrays
          ObjectSetup()
      }
}
```

Triangle Strip Decomposition

The 3DPRIM_TRISTRIP and 3DPRIM_TRISTRIP_REVERSE primitives specify a series of triangles arranged in a strip, as illustrated below.

3DPRIM_TRISTRIP[_REVERSE] Primitive



The decomposition process divides the strip into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v1,v2,v3; v2,v3,v4; and so on. Note that the *winding order* of the vertices alternates between CW (clockwise), CCW (counter-clockwise), CW, etc. The *provokingVertex* of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

The 3D pipeline uses the winding order of the vertices to distinguish between front-facing and back-facing triangles (Triangle Orientation (Face) Culling below). Therefore, the 3D pipeline must account for the alternation of winding order in strip triangles. The *invertOrientation* variable is generated and used for this purpose.

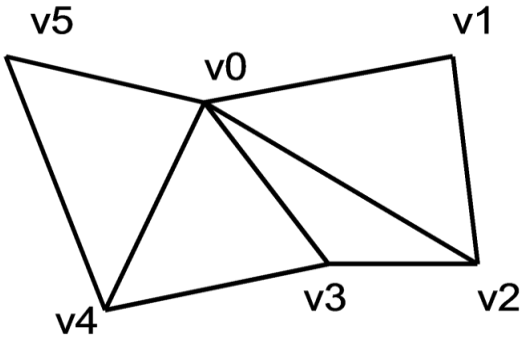
To accommodate the situation where the driver is forced to break an input strip primitive into multiple tristrip primitive commands (for example, due to ring or batch buffer size restrictions), two tristrip primitive types are supported. 3DPRIM_TRISTRIP is used for the initial section of a strip, and wherever a continuation of a strip starts with a triangle with a CW winding order. 3DPRIM_TRISTRIP_REVERSE is used for a continuation of a strip that starts with a triangle with a CCW winding order.

```
TriangleStripDecomposition()
   {
      objectType = 3DOBJ_TRIANGLE
      nV = 3
      provokingVtx = Triangle List/Strip Provoking Vertex Select
      if (primType == 3DPRIM_TRISTRIP)
         invertOrientation = FALSE
         else // primType == 3DPRIM_TRISTRIP_REVERSE
            invertOrientation = TRUE
      polyStippleEnable = TRUE
      for each (vertex I in [0..vertexCount-3])
        {
            v[0] arrays <- vIn[i] arrays
            v[1] arrays <- vIn[i+1] arrays
            v[2] arrays <- vIn[i+2] arrays
            ObjectSetup()
            invertOrientation = ! invertOrientation
        }
   }
```

Triangle Fan Decomposition

The 3DPRIM_TRIFAN and 3DPRIM_TRIFAN_NOSTIPPLE primitives specify a series of triangles arranged in a fan, as illustrated below.

3DPRIM_TRIFAN Primitive



The decomposition process divides the fan into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v0,v2,v3; v0,v3,v4; and so on. As there is no alternation in the vertex winding order, the *invertOrientation* variable is output as FALSE unconditionally. The *provokingVertex* of each object is taken from the **Triangle Fan Provoking Vertex** state variable, as programmed via SF_STATE.

Primitives of type 3DPRIM_TRIFAN_NOSTIPPLE are decomposed exactly the same way, except the *polyStippleEnable* variable is FALSE for the resulting objects being passed on to object setup. This will inhibit polygon stipple for these triangle objects.

```
TriangleFanDecomposition()
   {
       objectType = 3DOBJ_TRIANGLE
       nV = 3
       invertOrientation = FALSE
       provokingVtx = Triangle Fan Provoking Vertex Select
       if (primType == 3DPRIM_TRIFAN)
           polyStippleEnable = TRUE
           else // primType == 3DPRIM_TRIFAN_NOSTIPPLE
              polyStippleEnable = FALSE

       v[0] arrays <- vIn[0] arrays
         // the 1st vertex is common

       for each (vertex I in [1..vertexCount-2])
          {
              v[1] arrays <- vIn[i] arrays
              v[2] arrays <- vIn[i+1] arrays
              ObjectSetup()
          }
   }
```

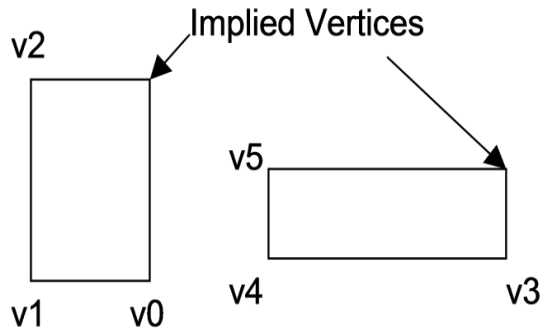
Polygon Decomposition

The 3DPRIM_POLYGON primitive is identical to the 3DPRIM_TRIFAN primitive with the exception that the *provokingVtx* is overridden with 0. This support has been added specifically for OpenGL support, avoiding the need for the driver to change the provoking vertex selection when switching between trifan and polygon primitives.

Rectangle List Decomposition

The 3DPRIM_RECTLIST primitive command specifies a list of independent, axis-aligned rectangles. Only the lower right, lower left, and upper left vertices (in that order) are included in the command - the upper right vertex is derived from the other vertices (in Object Setup).

3DPRIM_RECTLIST Primitive



The decomposition of the 3DPRIM_RECTLIST primitive is identical to the 3DPRIM_TRILIST decomposition, with the exception of the *objectType* variable.

```
RectangleListDecomposition()
   {
       objectType = 3DOBJ_RECTANGLE
       nV = 3
       invertOrientation = FALSE
       provokingVtx = 0
       for each (vertex I in [0..vertexCount-3] by 3)
          {
              v[0] arrays <-  vIn[i] arrays
              v[1] arrays <-  vIn[i+1] arrays
              v[2] arrays <-  vIn[i+2] arrays
              ObjectSetup()
          }
   }
```

Object Setup

The Object Setup subfunction of the SF stage takes the post-viewport-transform data associated with each vertex of a basic object and computes various parameters required for scan conversion. This includes generation of implied vertices, translations and adjustments on vertex positions, and culling (removal) of certain classes of objects. The final object information is passed to the Windower/Masker (WM) stage where the object is rasterized into pixels.

Invalid Position Culling (Pre/Post-Transform)

At input the the SF stage, any objects containing a floating-point NaN value for Position X, Y, Z, or RHW will be unconditionally discarded. Note that this occurs on an object (not primitive) basis.

If Viewport Transformation is enabled, any objects containing a floating-point NaN value for post-transform Position X, Y or Z will be unconditionally discarded.

Viewport Transformation

If the **Viewport Transform Enable** bit of SF_STATE is ENABLED, a viewport transformation is applied to each vertex of the object.

The VPIndex associated with the leading vertex of the object is used to obtain the **Viewport Matrix Element** data from the corresponding element of the SF_VIEWPORT structure in memory. For each object vertex, the following scale and translate transformation is applied to the position coordinates:

x' = **m00** * x + **m30**

y' = **m11** * y + **m31**

z' = **m22** * z + **m32**

Software is responsible for computing the matrix elements from the viewport information provided to it from the API.

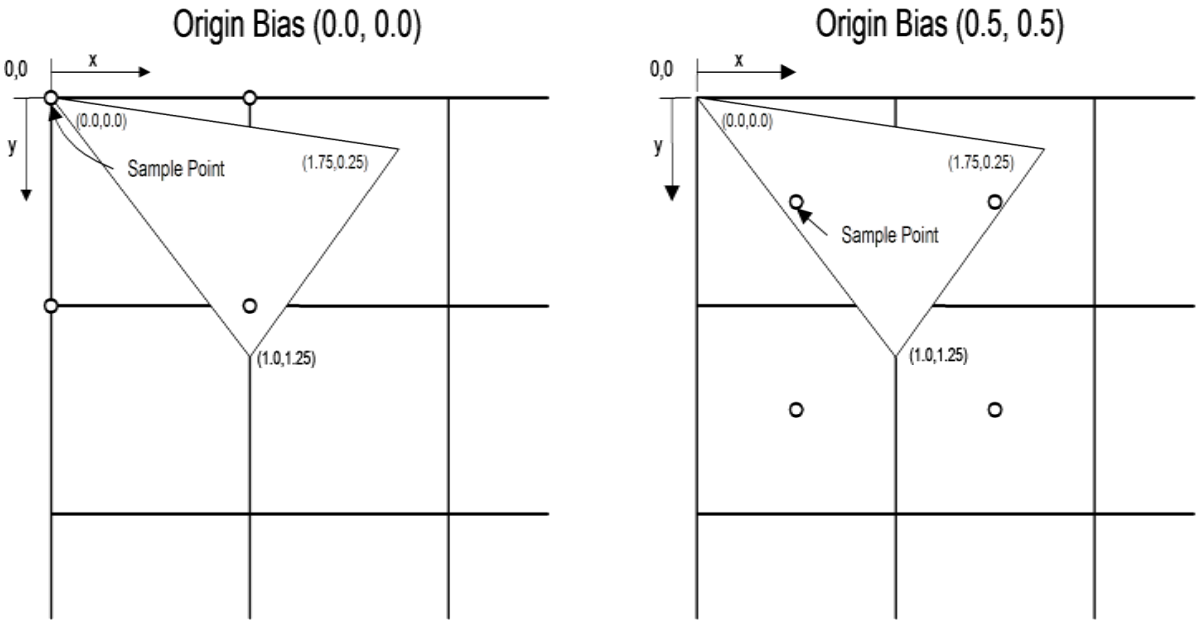
Destination Origin Bias

The positioning of the pixel sampling grid is programmable and is controlled by the **Destination Origin Horizontal/Vertical Bias** state variables (set via SF_STATE). If these bias values are both 0, pixels are sampled on an integer grid. Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0,0) falls within the primitive.

If the bias values are both 0.5, pixels are sampled on a "half" integer grid (i.e., X.5, Y.5). Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0.5,0.5) falls within the primitive. This positioning of the sample grid corresponds with the OpenGL rasterization rules, where "fragment centers" lay on a half-integer grid. It also corresponds with the Intel740 rasterizer (though that device did not employ "top left" rules).

Note that subsequent descriptions of rasterization rules for the various objects will be with reference to the pixel sampling grid.

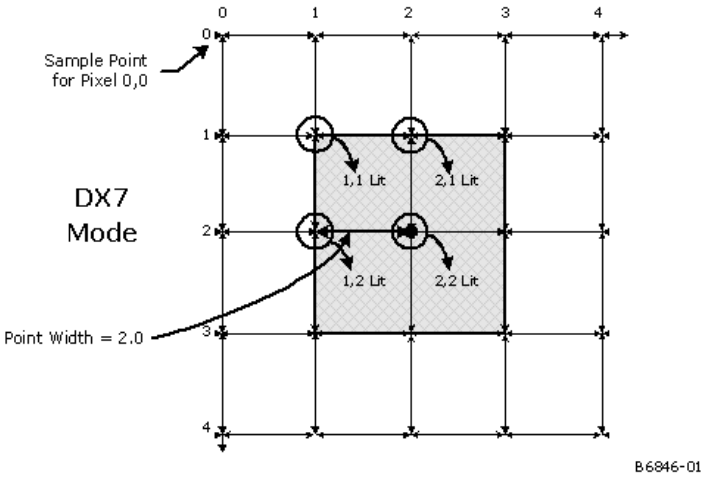
Destination Origin Bias



Point Rasterization Rule Adjustment

POINT objects are rasterized as square RECTANGLEs, with one exception: The **Point Rasterization Rule** state variable (in SF_STATE) controls the rendering of point object edges that fall directly on pixel sample points, as the treatment of these edge pixels varies between APIs.

RASTRULE_UPPER_LEFT



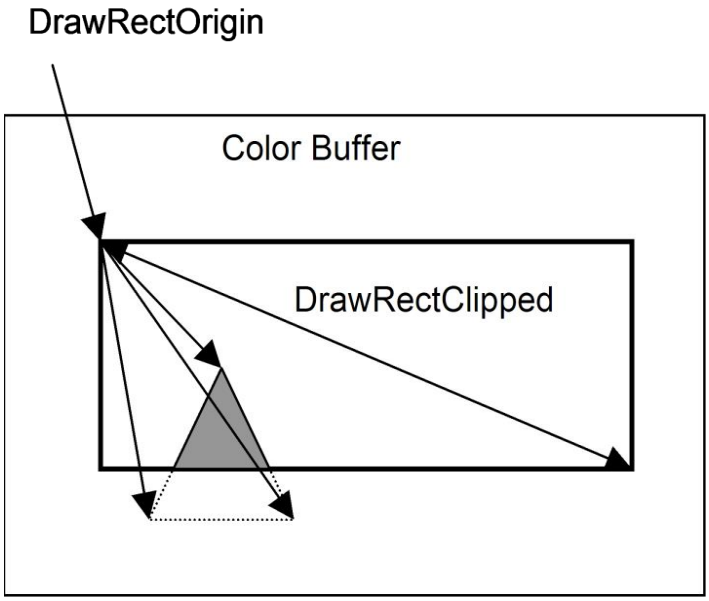
Drawing Rectangle Offset Application

The Drawing Rectangle Offset subfunction offsets the object's vertex X,Y positions by the pixel-exact, unclipped drawing rectangle origin (as programmed via the **Drawing Rectangle Origin X,Y** values in the 3DSTATE_DRAWING_RECTANGLE command). The Drawing Rectangle Offset subfunction (at least with respect to Color Buffer access) is unconditional, and therefore to (effectively) turn off the offset function

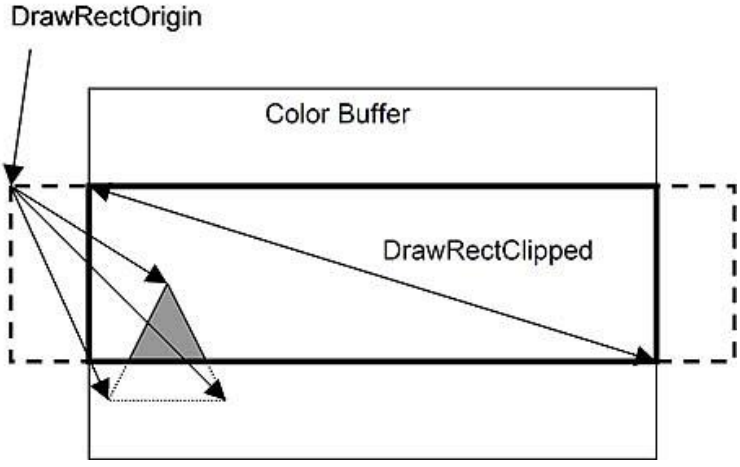
the origin would need to be set to (0,0). A non-zero offset is typically specified when window-relative or viewport-relative screen coordinates are input to the device. Here the drawing rectangle origin would be loaded with the absolute screen coordinates of the window's or viewport's upper-left corner.

Clipping of objects which extend outside of the Drawing Rectangle occurs later in the pipeline. Note that this clipping is based on the "clipped" draw rectangle (as programmed via the **Clipped Drawing Rectangle** values in the 3DSTATE_DRAWING_RECTANGLE command), which must be clamped by software to the rendertarget boundaries. The unclipped drawing rectangle origin, however, can extend outside the screen limits in order to support windows whose origins are moved off-screen. This is illustrated in the following diagrams.

Onscreen Draw Rectangle



Partially-offscreen Draw Rectangle



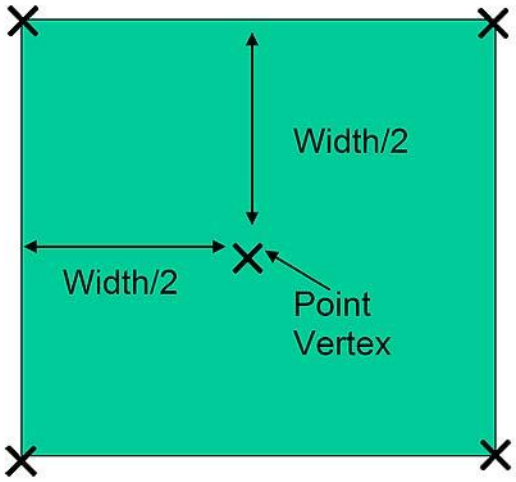
3DSTATE_DRAWING_RECTANGLE

Point Width Application

This stage of the pipeline applies only to 3DOBJ_POINT objects. Here the point object is converted from a single vertex to four vertices located at the corners of a square centered at the point's X,Y position. The width and height of the square are specified by a *point width* parameter. The **Point Width Source** value in SF_STATE determines the source of the point width parameter: the point width is either taken from the **Point Width** value programmed in SF_STATE or the PointWidth specified with the vertex (as read back from the vertex VUE earlier in the pipeline).

The corner vertices are computed by adding and subtracting one half of the point width.

Point Width Application

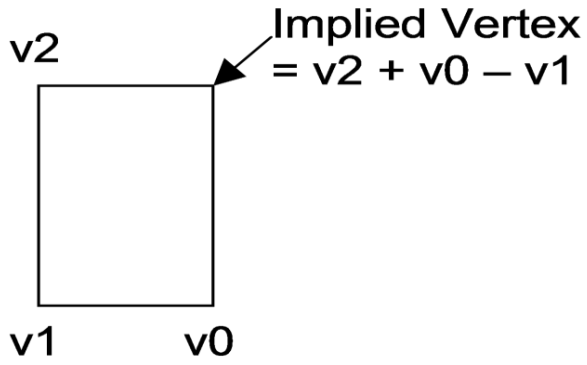


Z and W vertex attributes are copied from the single point center vertex to each of the four corner vertices.

Rectangle Completion

This stage of the pipeline applies only to 3DOBJ_RECTANGLE objects. Here the X,Y coordinates of the 4th (upper right) vertex of the rectangle object is computed from the first 3 vertices as shown in the following diagram. The other vertex attributes assigned to the implied vertex (v[3]) are UNDEFINED as they are not used. The Object Setup subfunction will use the values at only the first 3 vertices to compute attribute interpolants used across the entire rectangle.

Rectangle Completion



Vertex XY Clamping and Quantization

At this stage of the pipeline, vertex X and Y positions are in continuous screen (pixel) coordinates. These positions are quantized to subpixel precision by rounding the incoming values to the nearest subpixel (using round-to-nearest-or-even rules matching the DirectX reference device). The device supports rasterization with either 4 or 8 fractional (subpixel) position bits, as specified by the **Vertex SubPixel Precision Select** bit of SF_STATE.

The vertex X and Y screenspace coordinates are also *clamped* to the fixed-point "guardband" range supported by the rasterization hardware, as listed in the following table:

Per-Device Guardband Extents

Supported X,Y ScreenSpace "Guardband" Extent	Maximum Post-Clamp Delta (X or Y)
[-32K,32K-1]	N/A

Note that this clamping occurs after the Drawing Rectangle Origin has been applied and objects have been expanded (i.e., points have been expanded to squares, etc.). In almost all circumstances, if an object's vertices are actually modified by this clamping (i.e., had X or Y coordinates outside of the guardband extent the rendered object will not match the intended result. Therefore, software should take steps to ensure that this does not happen - e.g., by clipping objects such that they do not exceed these limits after the Drawing Rectangle is applied.

In addition, in order to be correctly rendered, objects must have a screenspace bounding box not exceeding 8K in the X or Y direction. This additional restriction must also be comprehended by software, i.e., enforced by use of clipping.

Degenerate Object Culling

At this stage of the pipeline, "degenerate" objects are discarded. This operation is automatic and cannot be disabled. (The object rasterization rules would by definition cause these objects to be "invisible" - this culling operation is mentioned here to reinforce that the device implementation optimizes these degeneracies as early as possible).

Degenerate Objects for definitions of degenerate objects.

Degenerate Objects

objType	Degenerate Object Definition
3DOBJ_POINT	Two or more corner vertices are coincident (i.e., the radius quantized to zero)
3DOBJ_LINE	The endpoints are coincident
3DOBJ_TRIANGLE	All three vertices are collinear, or any two vertices are coincident and SOLID fill mode applies to the triangle
3DOBJ_RECTANGLE	Two or more corner vertices are coincident

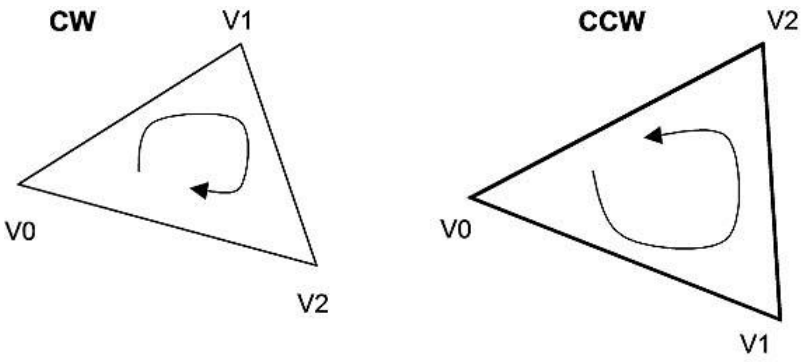
Triangle Orientation (Face) Culling

At this stage of the pipeline, 3DOBJ_TRIANGLE objects can be optionally discarded based on the "face orientation" of the object. This culling operation does not apply to the other object types.

This operation is typically called "back face culling", though front facing objects (or all 3DOBJ_TRIANGLE objects) can be selected to be discarded as well. Face culling is typically used to eliminate triangles facing away from the viewer, thus reducing rendering time.

The "winding order" of a triangle is defined by the triangle vertex's 2D (X,Y) screen space position when traversed from v0 to v1 to v2. That traversal proceeds in either a clockwise (CW) or counter-clockwise (CCW) direction. The "winding order" of a triangle is defined by the triangle vertex's 2D (X,Y) screen space position when traversed from v0 to v1 to v2. That traversal will proceed in either a clockwise (CW) or counter-clockwise (CCW) direction. A degenerate triangle is considered "backfacing", regardless of the FrontWinding state.

Triangle Winding Order



The **Front Winding** state variable in SF_STATE controls whether CW or CCW triangles are considered as having a "front-facing" orientation (at which point non-front-facing triangles are considered "back-facing"). The internal variable *invertOrientation* associated with the triangle object is then used to determine whether the orientation of a that triangle should be inverted. Recall that this variable is set in the Primitive Decomposition stage to account for the alternating orientations of triangles in strip primitives resulting form the ordering of the vertices used to process them.

The **Cull Mode** state variable in SF_STATE specifies how triangles are discarded according to their resultant orientation. See Degenerate Objects.

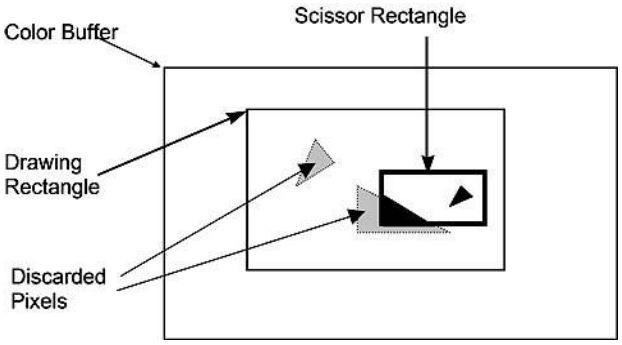
Cull Mode

CullMode	Definition
CULLMODE_NONE	The face culling operation is disabled.
CULLMODE_FRONT	Triangles with "front facing" orientation are discarded.
CULLMODE_BACK	Triangles with "back facing" orientation are discarded.
CULLMODE_BOTH	All triangles are discarded.

Scissor Rectangle Clipping

A *scissor* operation can be used to restrict the extent of rendered pixels to a screen-space aligned rectangle. If the scissor operation is enabled, portions of objects falling outside of the intersection of the scissor rectangle and the clipped draw rectangle are clipped (pixels discarded).

The scissor operation is enabled by the **Scissor Rectangle Enable** state variable in SF_STATE. If enabled, the VPIndex associated with the leading vertex of the object is used to select the corresponding SF_VIEWPORT structure. Up to 16 structures are supported. The **Scissor Rectangle X,Y Min,Max** fields of the SF_VIEWPORT structure defines a scissor rectangle as a rectangle in integer pixel coordinates relative to the (unclipped) origin of the Drawing Rectangle. The scissor rectangle is defined relative to the Drawing Rectangle to better support the OpenGL API. (OpenGL specifies the "Scissor Box" in window-relative coordinates). This allows instruction buffers with embedded Scissor Rectangle definitions to remain valid even after the destination window (drawing rectangle) moves.



Specifying either scissor rectangle xmin > xmax or ymin > ymax will cause all polygons to be discarded for a given viewport (effectively a null scissor rectangle).

Viewport Extents Test

Viewport extents test can be used to restrict the extent of rendered pixels to the viewport extents. If this operation is enabled, portion of the objects falling outside of the intersection of the scissor rectangle (if enabled) and the clipped draw rectangle and viewport extents are clipped (pixels discarded). This operation similar to the scissor test except both have different enables and the viewport extents can be programmed to the fractional float values.

This operation is enabled by the **View Transform Enable** state variable in **SF_STATE**. If enabled, the VPIndex associated with the leading vertex of the object is used to select the corresponding **SF_CLIP_VIEWPORT** structure. Up to 16 structures are supported. The X/Y **Min/Max ViewPort** fields of

the **SF_CLIP_VIEWPORT** structure defines viewport extents as a rectangle in float screen pixel coordinates relative to the (unclipped) origin of the Drawing Rectangle. Please note that these co-ordinates can be fractional values and hardware will do appropriate rounding and convert it to integer pixel co-ordinates (floor rounding used). This **View Transform Enable** state also controls the viewport transform so appropriate the viewport transform coefficients need to be populated in the SF_CLIP_VIEPWORT structure along with the viewport extents.

Final clip rectangle used to define the rendering area will now depend on three rectangles namely drawing rectangle, Scissor rectangle, Viewport Extents. If both **Scissor Rectangle Enable** and **View transform enable** are set then intersection of all rectangles (Viewport extents, Scissor rectangle, Draw rectangle) becomes final clip rectangle, while If only **Scissor Rectangle Enable** is enabled then the intersection of (Scissor rectangle, Draw rectangle) becomes final clip rectangle. If only **View transform enable** is enabled then intersection of (Viewport extents, Draw rectangle) become the final clip rectangle, while If none is enabled then (Draw rectangle) is the final clip rectangle.

Specifying either viewport extents xmin > xmax or ymin > ymax will cause all polygons to be discarded for a given viewport (effectively a null viewport).

Line Rasterization

The device supports three styles of line rendering: *zero-width* (*cosmetic*) lines, *non-antialiased* lines, and *antialiased* lines.

Non-antialiased lines are rendered as a polygon having a specified width as measured parallel to the major axis of the line. Antialiased lines are rendered as a rectangle having a specified width measured perpendicular to the line connecting the vertices.

The functions required to render lines are split between the SF and WM units. The SF unit is responsible for computing the overall geometry of the object to be rendered, including the pixel-exact bounding box, edge equations, etc., and therefore is provided with the screen-geometry-related state variables. The WM unit performs the actual scan conversion, determining the exact pixels included/excluded and coverage values for anti-aliased lines.

Zero-Width (Cosmetic) Line Rasterization

Note: The specification of zero-width line rasterization would be more correctly included in the WM Unit chapter, though is being included here to keep it with the rasterization details of the other line types.

When the **Line Width** is set to zero, the device will use special rules to rasterize zero-width ("cosmetic") lines. The **Anti-Aliasing Enable** state variable is ignored when **Line Width** is zero.

When the **LineWidth** is set to zero, the device will use special rules to rasterize "cosmetic" lines.

The rasterization rules also comply with the OpenGL conformance requirements (for 1-pixel wide non-smooth lines). Refer to the appropriate API specifications for details on these requirements.

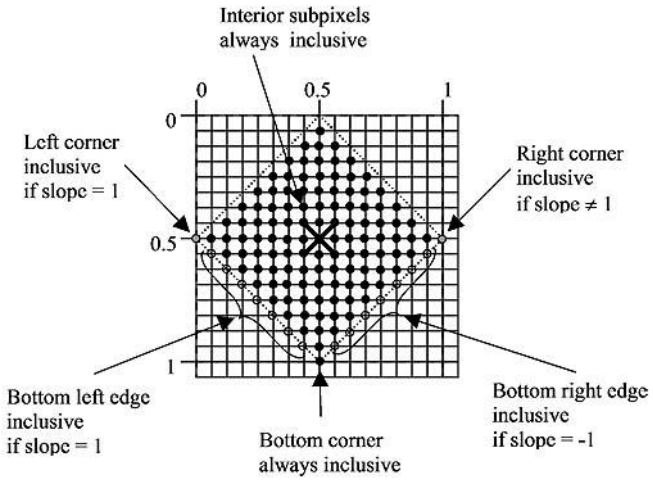
The GIQ rules basically intersect the directed, ideal line connecting two endpoints with an array of diamond-shaped areas surrounding pixel sample points. Wherever the line exits a diamond (including passing through a diamond), the corresponding pixel is lit. Special rules are used to define the subpixel locations that are considered interior to the diamonds, as a function of the slope of the line. When a line

ends in a diamond (and therefore does not exit that diamond), the corresponding pixel is not drawn. When a line starts in a diamond and exits that diamond, the corresponding pixel is drawn.

GIQ (Diamond) Sampling Rules - Legacy Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is ENABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.



The solid-colored subpixels are considered "interior" to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than 1/2.

The subpixels falling on the edges of the diamond (Manhattan distance = 1/2) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line slope is not exactly one, in which case the left corner subpixel is inclusive.** Including the right corner subpixel ensures that lines with slopes in the range (1, +infinity] or [-infinity, -1) touch a diamond even when they cross exactly between pixel diamonds. Including the left corner on slope=1 lines is required for proper handling of slope=1 lines (see (3) below) - where if the right corner was inclusive, a slope=1 line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
3. **The subpixels along the bottom left edge are inclusive only if the line slope = 1.** This is to correctly handle the case where a slope=1 line falls enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case:

One slope=1 line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would case the pixel to be lit twice - breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside" for slope=1 lines, we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.

4. **The subpixels along the bottom right edge are inclusive only if the line slope = -1.** Similar case as (3), except slope=-1 lines require the bottom right edge to be considered inclusive.

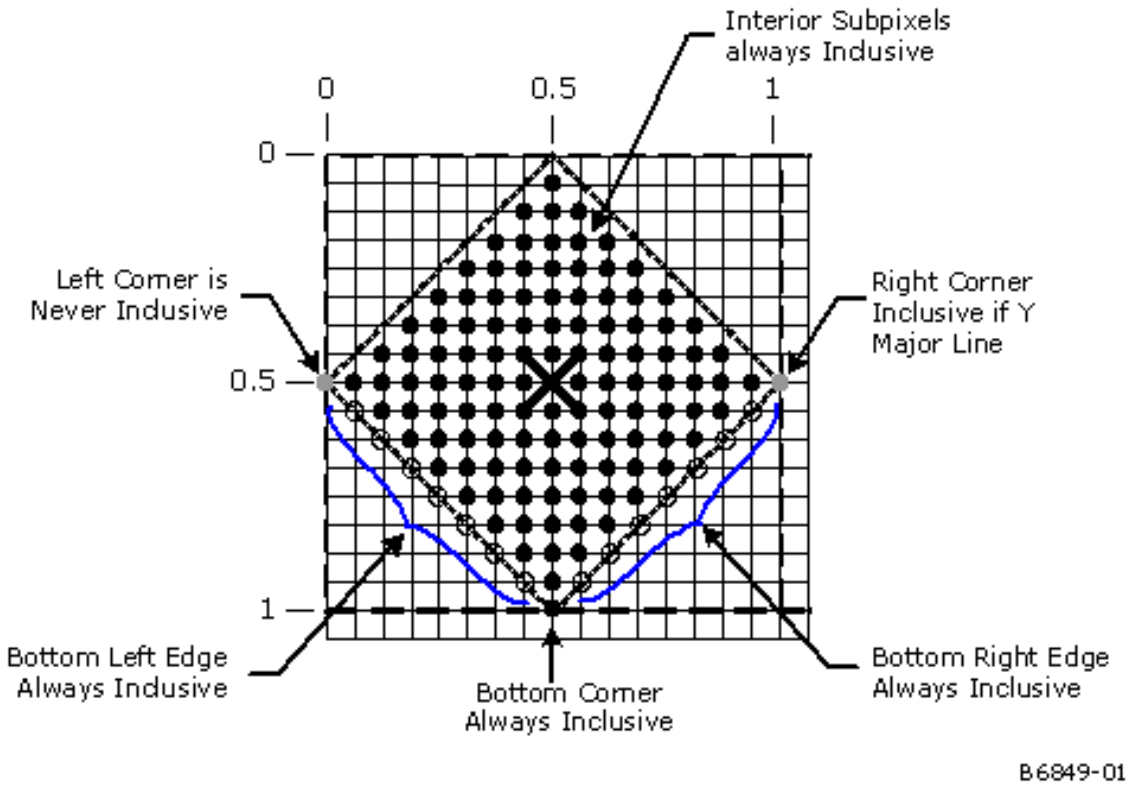
The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (slopePosOne, slopeNegOne).

```
delta_x                         = point.x - sample.x
delta_y                         = point.y - sample.y
distance                    = abs(delta_x) + abs(delta_y)
interior                    = (distance < 0.5)
bottom_corner               = (delta_x == 0.0) && (delta_y == 0.5)
left_corner                 = (delta_x == -0.5) && (delta_y == 0.0)
right_corner                = (delta_x == 0.5) && (delta_y == 0.0)
bottom_left_edge            = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge           = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)

inside = interior || bottom_corner || (slopePosOne ? left_corner : right_corner) ||
(slopePosOne && left_edge) || (slopeNegOne && right_edge)
```

GIQ (Diamond) Sampling Rules - DX10 Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is DISABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.



The solid-colored subpixels are considered "interior" to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than 1/2.

The subpixels falling on the edges of the diamond (Manhattan distance = 1/2) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line is not X Major (X Major is defined as -1 <= slope <= 1).** Including the right corner subpixel ensures that lines with slopes in the range (>1, +infinity] or [-infinity, <-1) touch a diamond even when they cross exactly between pixel diamonds.
3. **The left corner subpixel is never inclusive.** For Y Major lines, having the right corner subpixel as always inclusive requires that the left corner subpixel should never be inclusive, since a line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
4. **The subpixels along the bottom left edge are always inclusive.** This is to correctly handle the case where a line enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would cause the pixel to be lit twice - breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as

"inside", the pixel is only lit when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.

5. **The subpixels along the bottom right edge are always inclusive.** Same as case as (4), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (XMajor).

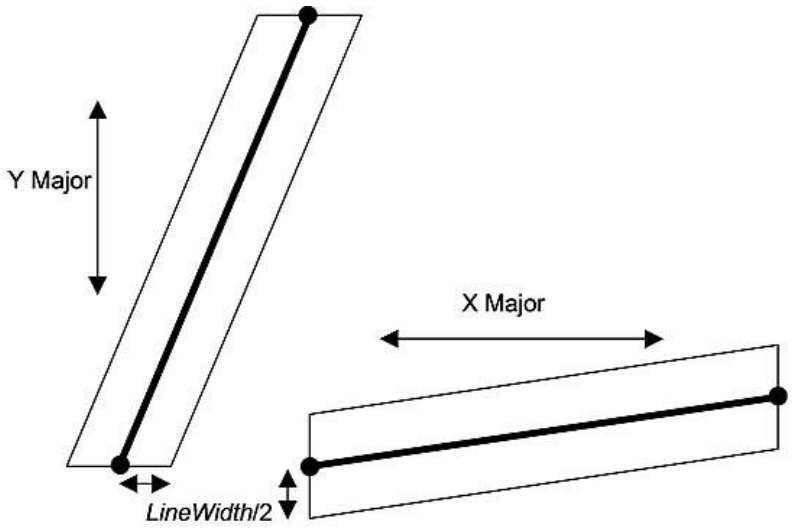
```
delta_x                         = point.x - sample.x
delta_y                         = point.y - sample.y
distance                        = abs(delta_x) + abs(delta_y)
interior                        = (distance < 0.5)
bottom_corner                           = (delta_x == 0.0)  && (delta_y == 0.5)
left_corner                     = (delta_x == -0.5) && (delta_y == 0.0)
right_corner                    = (delta_x == 0.5)  && (delta_y == 0.0)
bottom_left_edge                = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge               = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)

inside = interior || bottom_corner || (!XMajor && right_corner) || ( bottom_left_edge)
|| ( bottom_right_edge)
```

Non-Antialiased Wide Line Rasterization

Non-anti-aliased, non-zero-width lines are rendered as parallelograms that are centered on, and aligned to, the line joining the endpoint vertices. Pixels sampled interior to the parallelogram are rendered; pixels sampled exactly on the parallelogram edges are rendered according to the polygon "top left" rules.

The parallelogram is formed by first determining the major axis of the line (diagonal lines are considered x-major). The corners of the parallelogram are computed by translating the line endpoints by +/-(**Line Width** / 2) in the direction of the minor axis, as shown in the following diagram.

Non-Antialiased Line Rasterization

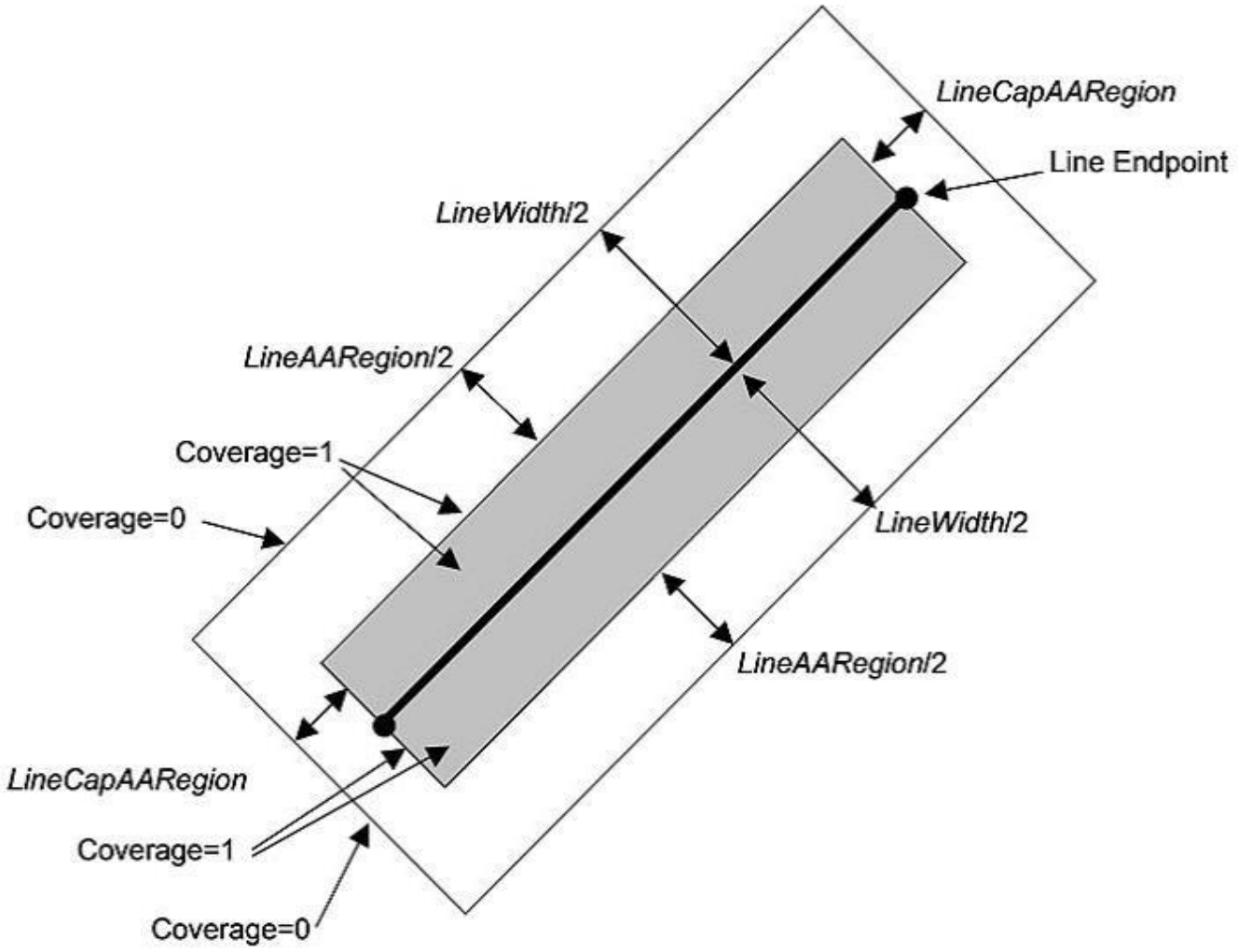


Anti-Aliased Line Rasterization

Anti-aliased lines are rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices. For each pixel in the rectangle, a fractional coverage value (referred to as Antialias Alpha) is computed - this coverage value is normally used to attenuate the pixel's alpha in the pixel shader thread. The resultant alpha value is therefore available for use in those downstream pixel pipeline stages to generate the desired effect (e.g., use the attenuated alpha value to modulate the pixel's color, and add the result to the destination color, etc.). Note that software is required to explicitly program the pixel shader and pixel pipeline to obtain the desired anti-aliasing effect - the device simply makes the coverage-attenuated pixel alpha values available for use in the pixel shader.

The dimensions of the rendered rectangle, and the parameters controlling the coverage value computation, are programmed via the **Line Width**, **Line AA Region**, and **Line Cap AA Region** state variables, as shown below. The edges parallel to the line are located at the distance (*LineWidth*/2) from the line (measured in screen pixel units perpendicular to the line). The end-cap edges are perpendicular to the line and located at the distance (*LineCapAARegion*) from the endpoints.

Anti-aliased Line Rasterization



Along the parallel edges, the coverage values ramp from the value 0 at the very edges of the rectangle to the value 1 at the perpendicular distance (*LineAARegion*/2) from a given edge (in the direction of the line). A pixel's coverage value is computed with respect to the closest edge. In the cases where (*LineAARegion*/2) < (*LineWidth*/2), this results in a region of fractional coverage values near the edges of the rectangle, and a region of "fully-covered" coverage values (i.e., the value 1) at the interior of the line. When (*LineAARegion*/2) == (*LineWidth*/2), only pixel sample points falling exactly on the line can generate fully-covered coverage values. If (*LineAARegion*/2) > (*LineWidth*/2), no pixels can be fully-covered (it is expected that this case is not typically desired).

Along the end cap edges, the coverage values ramp from the value 1 at the line endpoint to the value 0 at the cap edge - itself at a perpendicular distance (*LineCapAARegion*) from the endpoint. Note that, unlike the line-parallel edges, there is only a single parameter (*LineCapAARegion*) controlling the extension of the line at the end caps and the associated coverage ramp.

The regions near the corners of the rectangle have coverage values influenced by distances from both the line-parallel and end cap edges - here the two coverage values are multiplied together to provide a composite coverage value.

The computed coverage value for each pixel is passed through the Windower Thread Dispatch payload. The Pixel Shader kernel should be passed (unmodified) by the shader to the Render Cache as part of it's output message.

SF Pipeline State Summary

3DSTATE_RASTER

3DSTATE_RASTER

Signal	SF_INT::Multisample Rasterization Mode
Description	This field determines whether multisample rasterization is enabled and how pixel sample points are defined.
Formula	See Table: WM_INT::Multisample Rasterization Mode in 3D Pipeline Windower > Windower Pipelined State > 3DSTATE_WM > 3DSTATE_WM

Signal	SF_INT::Global Depth Offset Enable Solid
Description	This field determines when Global Depth bias gets enabled.
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Solid? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT:: Global Depth Offset Enable Wireframe
Description	This field determines when Global Depth bias gets enabled.
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Wireframe? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT:: Global Depth Offset Enable Point
Description	This field determines when Global Depth bias gets enabled.
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Point? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

3DSTATE_SF

The state used by the SF stage is defined by this inline state packet.

3DSTATE_SF

The SF Unit also receives 3DSTATE_RASTER. It also receives 3DSTATE_INT which is transparent to SW. 3DSTATE_INT provides 3DSTATE_WM, 3DSTATE_WM_HZ_OP, 3DSTATE_DETPH_BUFFER, and 3DSTATE_MULTISAMPLE fields.

Signal	SF_INT::Number of Multisamples
Description	Set the number of multisamples.
Formula	= (WM_INT::WM_HZ_OP) ? 3DSTATE_ WM_HZ_OP::Number of Multisamples : 3DSTATE_MULTISAMPLE::Number of Multisamples

Signal	SF_INT::Pixel Position Offset Enable
Description	Enables the device to offset pixel positions by 0.5 both in horizontal and vertical directions.
Formula	= (WM_INT::WM_HZ_OP) ? 3DSTATE_ WM_HZ_OP:: Pixel Position Offset Enable: 3DSTATE_MULTISAMPLE:: Pixel Position Offset Enable

Signal	SF_INT::Pixel Position Offset
Description	Causes the device to offset pixel positions by 0.5 both in horizontal and vertical directions. It is to be noted this is done to adjust the pixel co-ordinate system to DX9 like, so any screen space rectangles (eg: HiZ Clear, Resolve etc.) generated internally by driver in this mode needs to be aware of this offset adjustment and send the rectangles according to alignment restriction taking this offset adjustment into consideration.

Formula	= (SF_INT::Number of Multisamples >1) && (3DSTATE_MULTISAMPLE:: Pixel Location == PIXLOC_UL_CORNER) && SF_INT::Pixel Position Offset Enable

Signal	SF_INT: Global Depth Offset Enable Solid
Description	Set the number of multisamples
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Solid ? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT: Global Depth Offset Enable Wireframe
Description	Set the number of multisamples
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Wireframe ? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT: Global Depth Offset Enable Point
Description	Set the number of multisamples.
Formula	= (3DSTATE_RASTER:: Global Depth Offset Enable Point ? ((3DSTATE_RASTER::Global Depth Offset Constant != IEEE_FP_ZERO) || (3DSTATE_RASTER::Global Depth Offset Scale != IEEE_FP_ZERO)): Disable

Signal	SF_INT::FrontFace Fill Mode
Description	This state controls how front-facing triangle and rectangle objects are rendered.
Formula	= 3DSTATE_INT::WM_HZ_OP ? SOLID : 3DSTATE_RASTER:: FrontFace Fill Mode

Signal	SF_INT::BackFace Fill Mode
Description	This state controls how Back-facing triangle and rectangle objects are rendered.
Formula	= 3DSTATE_INT::WM_HZ_OP ? SOLID : 3DSTATE_RASTER:: BackFace Fill Mode

Signal	SF_INT::FrontWinding
Description	Determines whether a triangle object is considered "front facing" if the screen space vertex positions, when traversed in the order, result in a clockwise (CW) or counter-clockwise (CCW) winding order. Does not apply to points or lines.
Formula	= 3DSTATE_INT::WM_HZ_OP ? FRONTWINDING_CW : 3DSTATE_RASTER::FrontWinding

Signal	SF_INT::Cull Mode
Description	Controls removal (culling) of triangle objects based on orientation. The cull mode only applies to triangle objects and does not apply to lines, points or rectangles.
Formula	= SF_INT::WM_HZ_OP ? CULLMODE_BACK : 3DSTATE_RASTER:: Cull Mode

Signal	SF_INT::Scissor Rectangle Enable
Description	This field is used to control whether the Viewport Z extents (near, far) are considered in VertexClipTest.
Formula	= SF_INT::WM_HZ_OP ? 3DSTATE_WM_HZ_OP:: Scissor Rectangle Enable : 3DSTATE_RASTER::Scissor Rectangle Enable

Signal	SF_INT::Anti-aliasing Enable
Description	This field enables "alpha-based" line antialiasing.
Formula	= = SF_INT::WM_HZ_OP ? 3DSTATE_WM_HZ_OP:: Scissor Rectangle Enable : 3DSTATE_RASTER::Anti-aliasing Enable

Signal	SF_INT::Global Depth Offset Constant
Description	Specifies the constant term in the Global Depth Offset function.
Formula	= 3DSTATE_RASTER::Global Depth Offset Constant

Signal	SF_INT::Global Depth Offset Scale
Description	Specifies the constant term in the Global Depth Offset function.
Formula	= 3DSTATE_RASTER::Global Depth Offset Scale

Signal	SF_INT::Global Depth Offset Clamp
Description	Specifies the clamp term used in the Global Depth Offset function.
Formula	= 3DSTATE_RASTER::Global Depth Offset Clamp

Signal	SF_INT::Line Stipple Enable
Description	Specifies the clamp term used in the Global Depth Offset function.
Formula	= 3DSTATE_WM::Line Stipple Enable

Signal	SF_INT::RT Independent Rasterization Enable
Description	Enables RT Independent Rasterization.
Formula	= 3DSTATE_INT::WM_HZ_OP ? Disable : 3DSTATE_RASTER::ForcedSampleCount != NUMRASTSAMPLES_0

Signal	SF_INT::WM_HZ_OP
Description	Enables WM_HZ_OP.
Formula	= (3DSTATE_WM_HZ_OP::DepthBufferClear || 3DSTATE_WM_HZ_OP::DepthBufferResolve || 3DSTATE_WM_HZ_OP::Hierarchical Depth Buffer Resolve Enable || 3DSTATE_WM_HZ_OP::StencilBufferClear || 3DSTATE_WM_HZ_OP::StencilBufferResolve) ?

	Enable : Disable

Signal	SF_INT:: View Transform Enable
Description	Enables View Transform
Formula	= SF_INT::WM_HZ_OP ? Disable : 3DSTATE_SF::View Transform Enable

Signal	SF_INT::Render Target Array index
Description	Render Target Array index being render to
Formula	= 3DSTATE_INT::WM_HZ_OP ? 0 : Render Target Array index pipelined from clipper

Signal	SF_INT::Depth Buffer Surface Format
Description	Depth format being used
Formula	= 3DSTATE_INT:: Depth Buffer Surface Format

Signal	SF_INT::Viewport index
Description	Viewport being used.
Formula	= SF_INT::WM_HZ_OP ? 0 : Viewport index pipelined from clipper

SF_CLIP_VIEWPORT

The viewport-specific state used by both the SF and CL units (SF_CLIP_VIEWPORT) is stored as an array of up to 16 elements, each of which contains the DWords described below. The start of each element is spaced 16 DWords apart. The location of the first element of the array, as specified by both **Pointer to SF_VIEWPORT** and **Pointer to CLIP_VIEWPORT**, is aligned to a 64-byte boundary.

SCISSOR_RECT

Attribute Interpolation Setup

With the attribute interpolation setup function being implemented in hardware, a number of state fields in 3DSTATE_SF are utilized to control interpolation setup.

Number of SF Output Attributes sets the number of attributes that will be output from the SF stage, not including position. This can be used to specify up to 32, and may differ from the number of input attributes. The number of input attributes is derived from the **Vertex URB Entry Read Length** field. Note that this field is also used to specify whether swizzling is to be performed on Attributes 0-15 or Attributes 16-32. See the state field definition for details.

Attribute Swizzling

The first or last set of 16 attributes can be swizzled according to certain state fields. **Attribute Swizzle Enable** enables the swizzling for all 16 of these attributes, and each of the attributes has a 2-bit **Swizzle Select** field that controls swizzling with the following settings:

- INPUTATTR - This attribute is sourced from AttrInputReg[SourceAttribute].
- INPUTATTR_FACING - This attribute is sourced from AttrInputReg[SourceAttribute] if the object is front-facing, otherwise it is sourced from AttrInputReg[SourceAttribute+1].
- INPUTATTR_W - This attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination).
- INPUTATTR_FACING - If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination). If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1]. WYZW.

Each of the first or last set of 16 attributes also has a 5-bit **Source Attribute** field which specify, per output attribute (not component), which input attribute sources the output attribute when INPUTATTR is selected for **Swizzle Select**. A **Source Attribute** value of 0 corresponds to the 128-bit attribute immediately following the vertex 4D position. If INPUTATTR_FACING is selected, this specifies the first of two consecutive (front,back) input attributes, where the SourceAttribute value can be an odd or even number (just not 31, as that would place the back-face input attribute past the end of the input max complement of input attributes).

Constant overriding is also available for the first or last set of 16 attributes. Each attribute has a **Constant Source** field which specifies the constant values per swizzled attribute, with the following settings available:

- XYZW = 0000
- XYZW = 0001
- XYZW = 1111

Each channel of each attribute has a **Component Override** field to control whether the corresponding channel is overridden with the constant value defined in **Constant Source**.

Interpolation Modes

All 32 attributes have a **Constant Interpolation Enable** state field bit to specify whether all components of the post-swizzled attribute are to be interpolated as constant values (not varying over the pixels of the object). If set, the attribute at the provoking vertex is copied to a0, and a1 and a2 are set to zero - this

results in a constant interpolation of the provoking vertex value. If clear, the attribute is linearly interpolated. Attributes 0-15 are further subjected to Wrap Shortest processing on a per-component basis, via the **Attribute WrapShortest Enables** state bitfields. WrapShortest processing modifies the a1 and/or a2 values depending on attribute deltas. All

The table below indicates the output values of a0, a1, and a2 depending on interpolation mode settings.

	a0	a1	a2
Constant	A0	0.0	0.0
Linear	A0	A1-A0	A2-A0
Wrap Shortest	A0	(A1-A0)+1 (A1-A0) <= -0.5 (A1-A0)-1 (A1-A0) >= 0.5 (A1-A0) otherwise	(A2-A0)+1 (A2-A0) <= -0.5 (A2-A0)-1 (A2-A0) >= 0.5 (A2-A0) otherwise

Point Sprites

Normally all vertex attributes (including texture coordinates) other than position are simply replicated from the incoming point center vertex to the generated point object (corner) vertices. However, both DX9 and OGL support "sprite points", where some/all texture coordinates are replaced with full-scale 2D texture coordinates.

A 32-bit **PointSprite TextureCoordinate Enable** bit mask controls whether the corresponding vertex attribute is to be replaced by a sprite point texture coordinate. The global (not per-attribute) **Point Sprite TextureCoordinate Origin** field controls how the point object vertex (top/bottom, left/right) texture coordinates are generated:

UPPERLEFT	Left	Right
Top	(0,0,0,1)	(1,0,0,1)
Bottom	(0,1,0,1)	(1,1,0,1)

LOWERLEFT	Left	Right
Top	(0,1,0,1)	(1,1,0,1)
Bottom	(0,0,0,1)	(1,0,0,1)

The state used by "setup backend" is defined by the following inline state packet.

3DSTATE_SBE

The state used by "setup backend" is defined by the following inline state packet.

3DSTATE_SBE_SWIZ

SBE Unit also receives 3DSTATE_INT which is transparent to SW. 3DSTATE_INT provides 3DSTATE_VS, 3DSTATE_DS, and 3DSTATE_GS fields.

Signal	SBD_INT::Vertex URB Entry Read Lenth
Description	Specifies the amount of URB data read for each Vertex URB entry, in 256-bit register increments.
Formula	= (3DSTATE_SBE::Force Vertex URB Entry Read Length == Force) ? 3DSTATE_SBE::Vertex URB Entry Read Length : 3DSTATE_GS::GS_Enable ? 3DSTATE_GS::Vertex URB Entry Output Length : 3DSTATE_DS::DS_Enable ? 3DSTATE_DS::Vertex URB Entry Output Length : 3DSTATE_VS::Vertex URB Entry Output Length

Signal	SBE_INT::Vertex URB Entry Read Offset
Description	Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB
Formula	= (3DSTATE_SBE::Force Vertex URB entry Offset == Force) ? 3DSTATE_SBE::Vertex URB Entry Read Offset: 3DSTATE_GS::GS_Enable ? 3DSTATE_GS:: Vertex URB Entry Output Read Offset: 3DSTATE_DS:DS_Enable ? 3DSTATE_DS:: Vertex URB Entry Output Read Offset : 3DSTATE_VS:: Vertex URB Entry Output Read Offset

Signal	SBE_INT::PrimId_override
Description	When true indicates that SBE provides the Primitive ID.
Formula	= 3DSTATE_GS::GS_Enable ? false : 3DSTATE_SBE::Primitive ID Override Component Select !=0

Barycentric Attribute Interpolation

Given hardware clipper and setup, some of the previous flexibility in the algorithm used to interpolate attributes is no longer available. Hardware uses barycentric parameters to aid in attribute interpolation, and these parameters are computed in hardware per-pixel (or per-sample) and delivered in the thread payload to the pixel shader. Also delivered in the payload are a set of vertex deltas (a0, a1, and a2) per channel of each attribute.

There are six different barycentric parameters that can be enabled for delivery in the pixel shader payload. These are enabled via the **Barycentric Interpolation Mode** bits in 3DSTATE_WM.

In the pixel shader kernel, the following computation is done for each attribute channel of each pixel/sample given the corresponding attribute channel a0/a1/a2 and the pixel/sample's b1/b2 barycentric parameters, where A is the value of the attribute channel at that pixel/sample:

A = a0 + (a1 * b1) + (a2 * b2)

Depth Offset

The state for depth offset in 3DSTATE_SF controls the depth offset function. Since this function was previously contained in the Windower stage, refer to the "Depth Offset" section in the Windower chapter for more details on this function.

Other SF Functions

The only other SF-related function is statistics gathering.

Statistics Gathering

The SF stage itself does not have any associated pipeline statistics; however, it counts the number of objects being output by the clipper on the clipper's behalf, since it is less feasible to have the CLIP unit figure out how many objects have been output by a clip thread. It is easy for the SF unit to count the number of objects it receives from the CLIP stage since it is decomposing the output primitive topologies into objects anyway.

If the **Statistics Enable** bit is set in SF_STATE, then SF will increment the CL_PRIMITIVES_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) once for each object in each primitive topology it receives from the CLIP stage. This bit should always be set if clipping is enabled and pipeline statistics are desired.

Software should always clear the **Statistics Enable** bit in SF_STATE if the clipper is disabled since objects SF receives are not considered "primitives output by the clipper" unless the clipper is enabled. Note that the clipper can be disabled either using bypass mode via a PIPELINE_STATE_POINTERS command with **Clip Enable** clear *or* by setting **Clip Mode** in CLIP_STATE to CLIPMODE_ACCEPT_ALL.

Tile Based Immediate Mode Rendering

The contemporary tile-based GPU architectures follow the two pass approach where the first pass does the sorting of the geometry into screen tiles and second pass does the pixel processing taking one tile at a time to get the band width benefits. Tile Based Immediate Mode Rendering (TBIMR) adds the optional ability to implement the tiling by storing a batch primitives on die and replaying that batch one screen tile at a time backed up by a tile cache. Unlike the traditional tile-based architectures this only requires single pass to implement the tiling and rest of complexity is implemented in the hardware.

TBIMR Programming Model

Programming the TBIMR will work just like programming any Render+ POSH pipeline with few additional commands and fields programmed exclusively to enable the feature. Unlike the PTBR, SW is not required to loop the tile batch buffers over all the tiles.

There are two main aspects that SW needs to setup to enable the tiling using this feature. Firstly, it needs to setup dimensions of the screen space tile based on parameters like render target resolution, render

target formats, number of multi samples etc. This also depends on the implementation specific parameters like unified tile cache size and maximum number of tiles supported. The algorithm for evaluating the tile dimension and number of tiles in horizontal and vertical direction is same as that of PTBR. These dimensions and number of tiles are programmed to the hardware through the 3DSTATE_TBIMR_TILE_PAS_INFO command. HW will evaluate the tile dimensions from the origin based on these parameters. If both the horizontal and vertical tile count is zero, then tiling is not enabled and will work in legacy mode. The expectation is that 3DSTATE_TBIMR_TILE_PAS_INFO will be programmed to the hardware on every render pass or whenever there is change in the tile dimensions or counts.

Secondly SW needs to turn on the TBIMR on every drawcall by setting the bits in the 3DPRIMITIVE command. SW is free to set/reset this bit on every drawcall based on heuristics and hardware will take care of appropriately enabling and disabling the tiling for those draw calls. It is to be noted that currently there is no restriction on enabling this feature based on primitive type and for maximum benefit it must be turned on for all the primitive types.

Along with this L3 cache configuration needs to be programmed appropriately to allocate the unified tile cache or Specific allocation for C/Z streams separately.

Parameter	Value
Unified Tile Cache Size per Slice	512 KB
Number of Tile supported	1024

Hardware implicitly closes the batch if there is any top of the pipe blocking events that requires it to close the batch and process all the stores polygons through the pixel pipe. These events include stalling flushes, depth statistics queries etc. There is also optional capability added in the PIPE_CONTROL command to explicitly force the closure of the batch, this is equivalent to end of tile pass indication that SW can use to close the batch

Pixel Hashing Mode

The Slice Hashing table is used to assign the pixels of the render target to a slice. The table has 16x16 elements and each element in the table is map to a slice. Each cell maps to a 16x16 pixels or 32x32 pixels (programable by cross slice hashing mode).

The current default table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
3	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
4	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
5	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
6	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
7	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6
8	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
9	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8
10	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
11	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10
12	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
13	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
14	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
15	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

The table is replicated until it covers the render target.

In the pixel hashing mode each TBIMR tile contains one or more elements of the Slice Hashing Table. The size of the TBIMR tile must be a multiple of the Slice Hashing table element size.

The slice hashing table that decides where to send each poly independent of the tile.

Example:

Tile rectangle width is 160 pixels (10 Mega spans 10x16)

Tile rectangle height is 96 pixels (6 Mega spans 6x16)

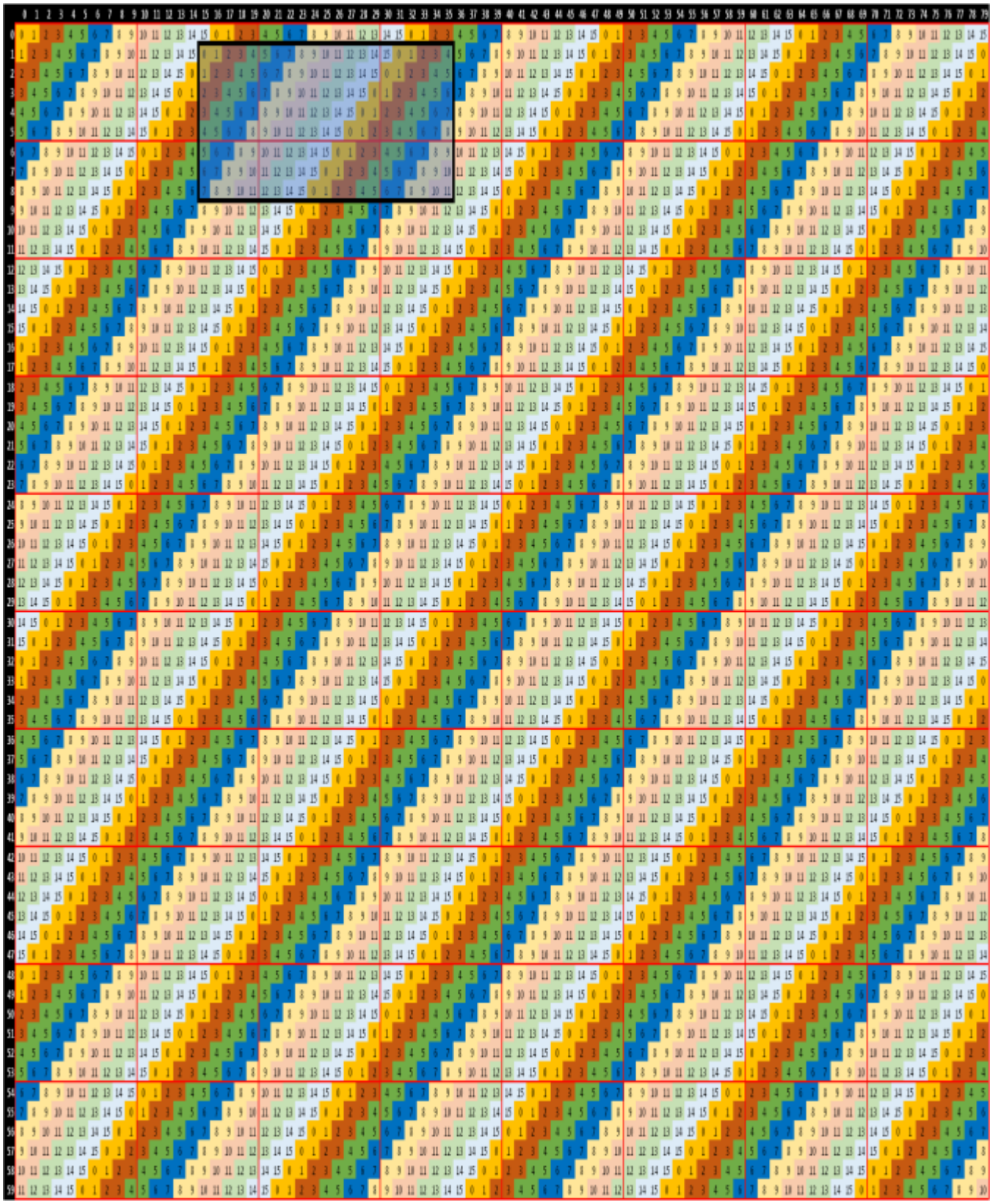
Horizontal Tile Count is 8

Vertical Tile Count is 10

A poly that has a bounding box from pixels (243, 25) to (573, 141)

It has a mega span bounding box (15, 1) to (35, 8)

The following figure shows the mapping of all the mega spans to the slices and the bounding box. The slice hashing table is replicated until it covers the entire render target.



The tiles used are Tile (0, 1), Tile (0, 2), Tile (0,3), Tile (1,1), Tile (1, 2) and Tile (1,3)

Slice Hashing Table

Slice Hashing

Table based Hashing

Slice_id hash is a lookup into a 256-entry slice_hash_table. The lowest 4 bits of the pixel block X,Y is used to index into the 16x16 table.

If (16x16 hashing)

X[3:0] = x[7:4]

Y[3:0] = y[7:4]

Else if (32x32 hashing)

X[3:0] = x[8:5]

Y[3:0] = y[8:5]

endif

Hz Slice = Slice Hash Table[Y][X] % NUM_HZ_IN_A_SLICE

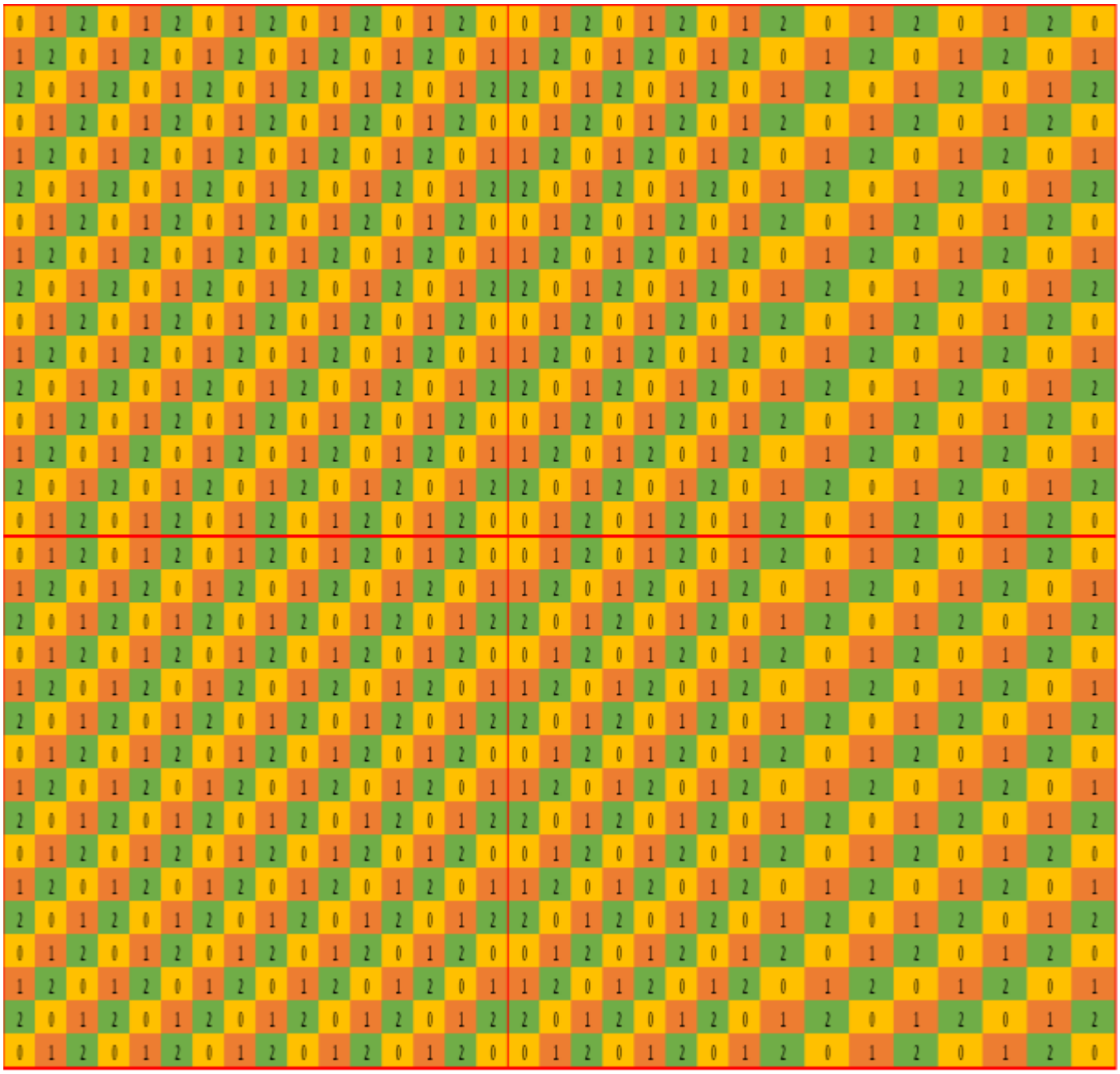
Default ROM Table

When **Slice Hashing Table Enable** is set to false, slice_hash_table defaults to ROM tables based on the current number of Z Pipe Id, and the following effective slice_id.

Current default hash table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
1	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1
2	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
4	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1
5	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
6	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
7	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1
8	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
9	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
10	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1
11	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
12	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
13	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1
14	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
15	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0

If we replicate the table:



In case of shutdown Hz Pipes.

If the list of enabled Hz pipes is pm_mode_sliceen

The following algorithm shows how to create the default table:

Enabled_Z <-- 0

For i from 0 to 15

While (not pm_mode_sliceen[Enabled_Z])

Enabled_Z <-- (Enabled_Z + 1) % GT_SC_Z_PER_SLICE

End while

Physical_array[i] <-- Enabled_Z

Enabled_Z <-- (Enabled_Z + 1) % GT_SC_Z_PER_SLICE

End For

For y from 0 to 15

For x from 0 to 15

Slice Hash Table[Y][X] <-- Physical_array[(y+x) % NUM_SUBSLICES]

End For

End For

For Example:

In this example pm_mode_sliceenis: 0x5

Physical_array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2

The table is:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
1	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0
2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
3	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0
4	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
5	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0
6	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
7	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0
8	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
9	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0
10	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
11	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0
12	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
13	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0
14	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
15	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0

This only a default function programming table, but for performance it needs to be optimized and reprogrammed based on the shutdown scenarios.

3DSTATE_SLICE_TABLE_STATE_POINTERS

SLICE_HASH_TABLE - SLICE_HASH_TABLE

Slice Hashing

Table based Hashing

Slice_id hash is a lookup into a 256 entry slice_hash_table. The lowest 4 bits of the pixel block X,Y is used to index into the 16x16 table.

If (16x16 hashing)

X[3:0] = x[7:4]

Y[3:0] = y[7:4]

Else if (32x32 hashing)

X[3:0] = x[8:5]

Y[3:0] = y[8:5]

endif

Slice = Slice Hash Table[Y][X] / NUM_HZ_IN_A_SLICE

Hz Slice = Slice Hash Table[Y][X] % NUM_HZ_IN_A_SLICE

Default ROM Table

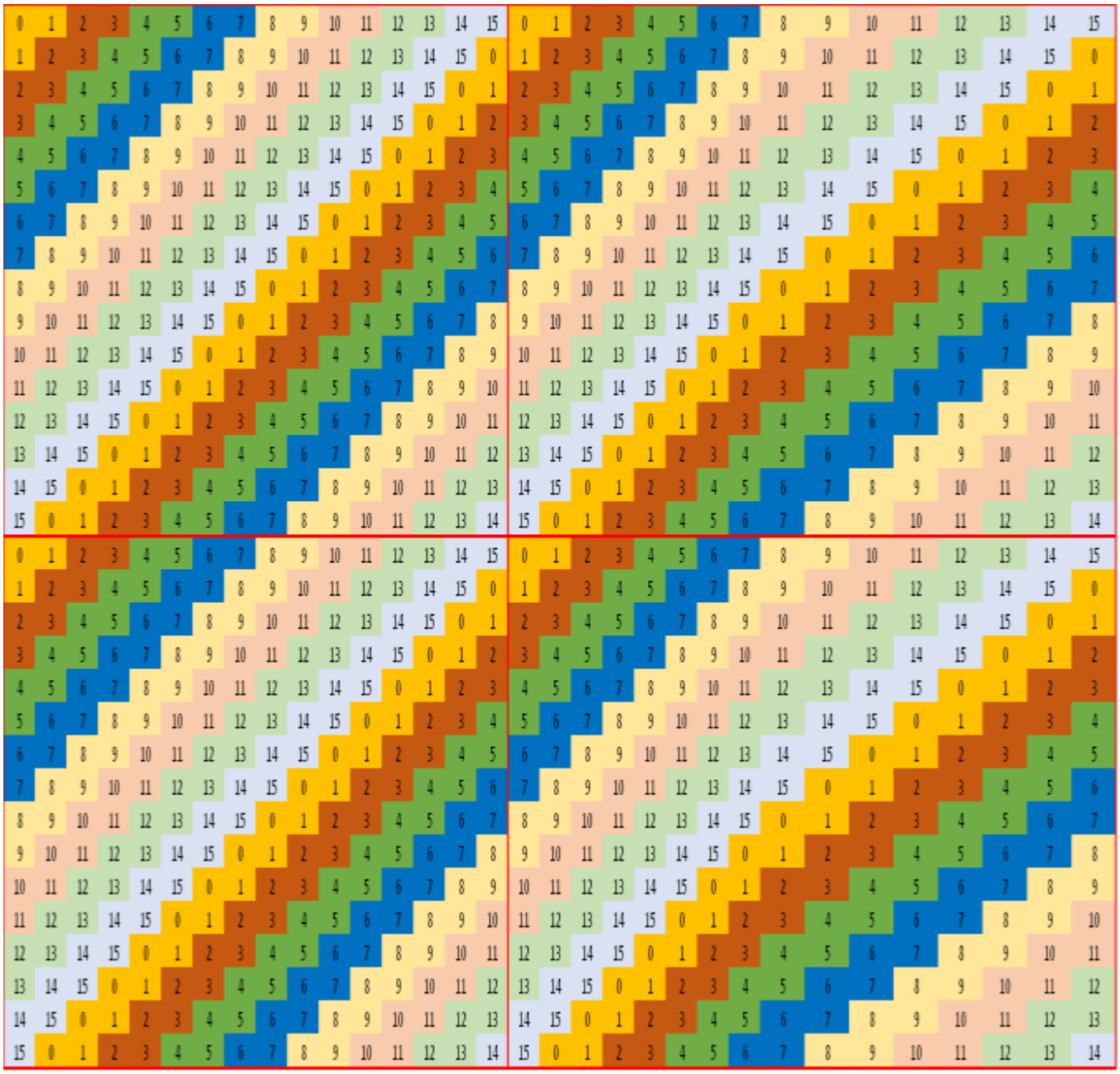
When **Slice Hashing Table Enable** is set to false, slice_hash_table defaults to ROM tables based on the current number of Z Pipe Id, and the following effective slice_id.

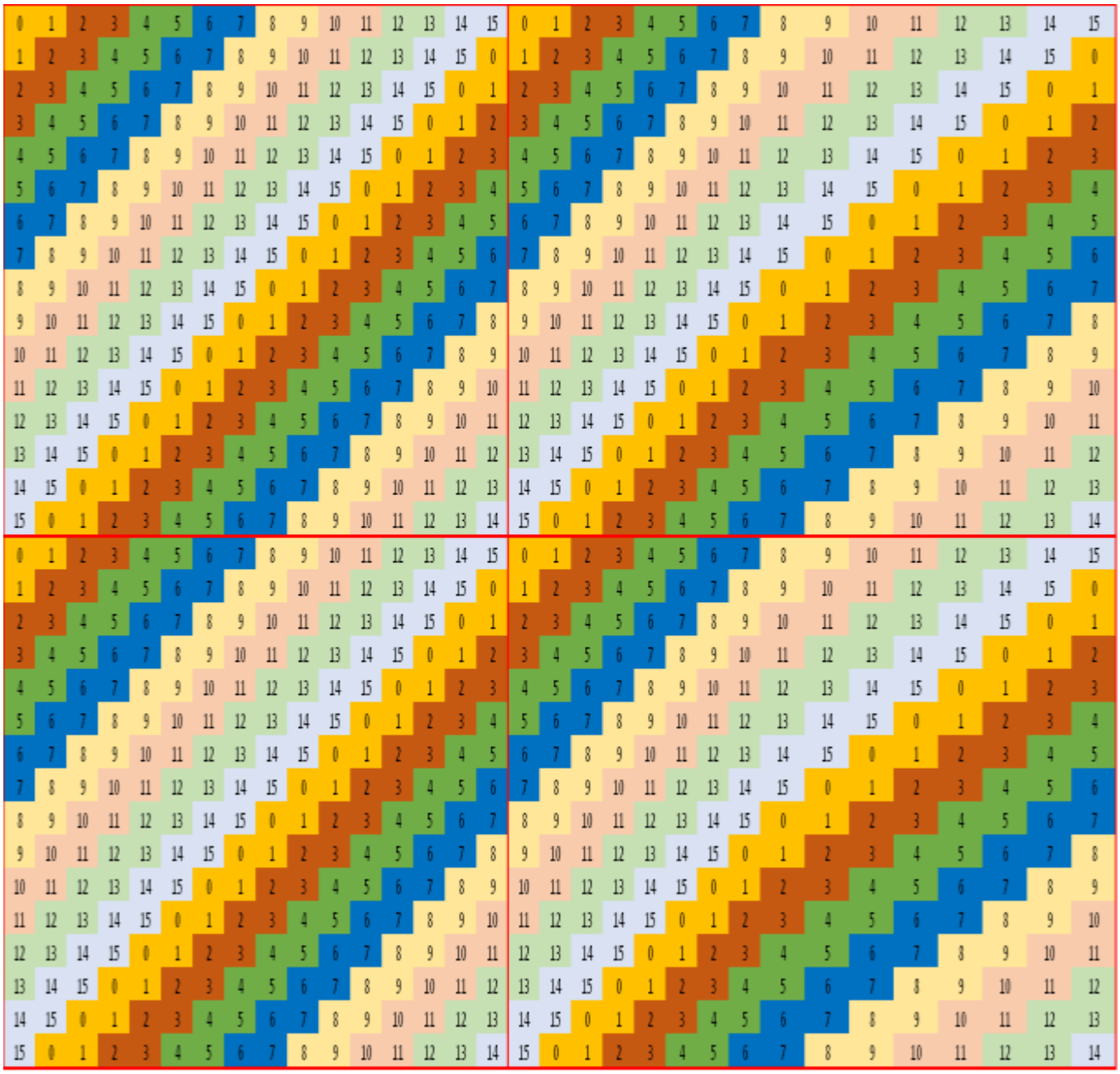
Current default hash table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
3	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
4	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
5	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
6	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
7	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6
8	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
9	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8
10	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
11	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10
12	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
13	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
14	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
15	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
3	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
4	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
5	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
6	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
7	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6
8	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
9	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8
10	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
11	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10
12	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
13	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
14	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
15	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If we replicate the table:


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1
3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2
4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3
5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4
6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5
7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6
8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7
9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8
10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9
11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10
12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12
14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13
15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14



In case of shutdown Hz Pipes.

A Hz Pipe is shutdown if both DSS are disabled.

The following algorithm defines a default mode, but it splits the render target into all enabled Hz Pipes without considering the number of DSS/Hz.

Only the Fuse Configuration is used for calculating Z-Pipe hashing and Dynamic Slice Shutdown has no effect.

If the list of enabled Hz pipes is pm_mode_dualsubsliceen

The following algorithm shows how to create the default table:

NUM_SUBSLICES = GT_NUM_SLICES*GT_SC_Z_PER_SLICE

Enabled_Z <-- 0

For i from 0 to 15

While (not pm_mode_sliceen[Enabled_Z/GT_NUM_GSLICES][Enabled_Z % GT_SC_Z_PER_SLICE])

Enabled_Z <-- (Enabled_Z + 1) % NUM_SUBSLICES

End while

Physical_array[i] <-- Enabled_Z

Enabled_Z <-- (Enabled_Z + 1) % NUM_SUBSLICES

End For

For y from 0 to 15

For x from 0 to 15

Slice Hash Table[Y][X] <-- Physical_array[(y+x) % NUM_SUBSLICES]

End For

End For

For Example:

In this example pm_mode_sliceen is: 0x4063

Physical_array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0

The table is:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0
1	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0	0
2	5	6	14	0	1	5	6	14	0	1	5	6	14	0	0	1
3	6	14	0	1	5	6	14	0	1	5	6	14	0	0	1	5
4	14	0	1	5	6	14	0	1	5	6	14	0	0	1	5	6
5	0	1	5	6	14	0	1	5	6	14	0	0	1	5	6	14
6	1	5	6	14	0	1	5	6	14	0	0	1	5	6	14	0
7	5	6	14	0	1	5	6	14	0	0	1	5	6	14	0	1
8	6	14	0	1	5	6	14	0	0	1	5	6	14	0	1	5
9	14	0	1	5	6	14	0	0	1	5	6	14	0	1	5	6
10	0	1	5	6	14	0	0	1	5	6	14	0	1	5	6	14
11	1	5	6	14	0	0	1	5	6	14	0	1	5	6	14	0
12	5	6	14	0	0	1	5	6	14	0	1	5	6	14	0	1
13	6	14	0	0	1	5	6	14	0	1	5	6	14	0	1	5
14	14	0	0	1	5	6	14	0	1	5	6	14	0	1	5	6
15	0	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0
1	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0	0
2	5	6	14	0	1	5	6	14	0	1	5	6	14	0	0	1
3	6	14	0	1	5	6	14	0	1	5	6	14	0	0	1	5
4	14	0	1	5	6	14	0	1	5	6	14	0	0	1	5	6
5	0	1	5	6	14	0	1	5	6	14	0	0	1	5	6	14
6	1	5	6	14	0	1	5	6	14	0	0	1	5	6	14	0
7	5	6	14	0	1	5	6	14	0	0	1	5	6	14	0	1
8	6	14	0	1	5	6	14	0	0	1	5	6	14	0	1	5
9	14	0	1	5	6	14	0	0	1	5	6	14	0	1	5	6
10	0	1	5	6	14	0	0	1	5	6	14	0	1	5	6	14
11	1	5	6	14	0	0	1	5	6	14	0	1	5	6	14	0
12	5	6	14	0	0	1	5	6	14	0	1	5	6	14	0	1
13	6	14	0	0	1	5	6	14	0	1	5	6	14	0	1	5
14	14	0	0	1	5	6	14	0	1	5	6	14	0	1	5	6
15	0	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14

This only a default function programming table, but for performance it needs to be optimized and reprogrammed based on the shutdown scenarios.

Programmable Hashing

When **Slice Hashing Table Enable** is set to true, slice_hash_table is fetched via indirect state. An array of slice_hash_tables is stored at memory pointed to by **Slice Hash Table State Pointer**. All Slice Hashing Tables should have the same values. No slice_hash_table is fetched when there is only one active slice.

DSTATE_SLICE_TABLE_STATE_POINTERS

Windower (WM) Stage

Overview

As mentioned in the *SF Unit* chapter, the SF stage prepares an object for scan conversion by the Window/Masker (WM) unit Refer to the *SF Unit* chapter for details on the screen-space geometry of objects to be rendered The WM unit uses the parameters provided by the SF unit in the object-specific rasterization algorithms.

The WM stage of the 3D pipeline performs the following operations (at a high level)

- Pre-scan-conversion modification of some primitive attributes, including
    - Application of Depth Offset to the position Z attribute
- Scan-conversion of the various primitive types, including
    - 2D clipping to the scissor/draw rectangle intersection
- Spawning of Pixel Shader (PS) threads to process the pixels resulting from scan-conversion

The spawned Pixel Shader (PS) threads are responsible for the following (high-level) operations

- interpolation of vertex attributes (other than X,Y,Z) to the pixel location
- performing any "Pixel Shader" operations dictated by the API PS program
    - Using the Sampler shared function to sample data from "texture" surfaces
    - Using the DataPort to perform general memory I/O
- Submitting the shaded pixel results to the DataPort for any subsequent "blending" (aka Output Merger) operation and write to the RenderCache.

The WM unit keeps a scoreboard of pixels being processed in outstanding PS threads in order to guarantee in-order rasterization results This allows the WM unit to overlap processing of several objects.

Inputs from SF to WM

The outputs from the SF stage to the WM stage are mostly comprised of implementation-specific information required for the rasterization of objects The types of information is summarized below, but as the interface is not exposed to software a detailed discussion is not relevant to this specification.

- PrimType of the object
- VPIndex, RTAIndex associated with the object
- Handle of the Primitive URB Entry (PUE) that was written by the SF (Setup) thread This handle will be passed to all WM (PS) threads spawned from the WM's rasterization process.
- Information regarding the X,Y extent of the object (e.g., bounding box, etc.)
- Edge or line interpolation information (e.g., edge equation coefficients, etc.)

- Information on where the WM is to start rasterization of the object
- Object orientation (front/back-facing)
- Last Pixel indication (for line drawing)

Windower Pipelined State

3DSTATE_WM

The following inline state packets define the state used by the windower stage for different generations.

State Packets
3DSTATE_WM

Programming Note	
Context:	Windower Pipelined State
Note: WM Unit also receives 3DSTATE_WM_HZ_OP, 3DSTATE_RASTER, 3DSTATE_MULTISAMPLE, 3DSTATE_WM_CHROMAKEY, 3DSTATE_PS_BLEND, and 3DSTATE_PS_EXTRA	

Signal	WM_INT:: Pixel Shader Computed Stencil
Description	This field specifies the computed stencil mode for the pixel shader.
Formula	= (WM_INT::WM_HZ_OP) ? 0 : 3DSTATE_PS_EXTRA::Computed Stencil

Signal	WM_INT::ThreadDispatchEnable
Description	This bit, if set, indicates that it is possible for a PS thread to modify a render target.
Formula	= (3DSTATE_WM::ForceThreadDispatch == ON) || ((3DSTATE_WM::ForceThreadDispatch != OFF) && ! WM_INT::WM_HZ_OP && 3DSTATE_PS_EXTRA::PixelShaderValid && ((!3DSTATE_PS_EXTRA::PixelShaderDoesNotWriteRT && 3DSTATE_PS_BLEND::HasWriteableRT) || (3DSTATE_PS_EXTRA::PixelShaderHasUAV)

	|| WM_INT:: Pixel Shader Kill Pixel || (WM_INT::Pixel Shader Computed Depth Mode != PSCDEPTH_OFF && (WM_INT::Depth Test Enable || WM_INT::Depth Write Enable)) || (3DSTATE_PS_EXTRA::Computed Stencil && WM_INT::Stencil Test Enable) || (3DSTATE_WM::EDSC_Mode == 1 && (WM_INT::Depth Test Enable || WM_INT::Depth Write Enable || WM_INT::Stencil Test Enable)) || (WM_INT::RT Independent Rasterization Enable)))

Signal	WM_INT::Pixel Shader Computed Depth Mode
Description	This field specifies the computed depth mode for the pixel shader.
Formula	= (3DSTATE_PS_EXTRA::ForceComputedDepth == Force) ? 3DSTATE_PS_EXTRA::Pixel Shader Computed Depth Mode : (WM_INT::WM_HZ_OP || WM_INT::RT Independent Rasterization Enable) ? PSCDEPTH_OFF: 3DSTATE_PS_EXTRA::Pixel Shader Computed Depth Mode

Signal	WM_INT::Pixel Shader Uses Source Depth
Description	This bit, if ENABLED, indicates that the PS kernel requires the source depth value (vPos.z) to be passed in the payload.
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Source Depth

Signal	WM_INT::Pixel Shader Uses Source W
Description	This bit, if ENABLED, indicates that the PS kernel requires the interpolated source W value (vPos.w) to be passed in the payload
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Source W

Signal	WM_INT::Pixel Shader Uses Input Coverage Mask
Description	This bit, if ENABLED, indicates that the PS kernel requires the input coverage mask to be passed in the payload.
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Uses Input Coverage Mask

Signal	WM_INT::Multisample Rasterization Mode
Description	This field determines whether multisample rasterization is enabled and how pixel sample points are defined.
Formula	See Table below: WM_INT::Multisample Rasterization Mode

WM_INT::Multisample Rasterization Mode

3DSTATE_RASTER:: Force Multisampling	Force	Force	Force	Force	Normal	Normal	Normal
3DSTATE_RASTER:: DX Multisample Rasterization Mode	MSRASTMODE_ OFF_PIXEL	MSRASTMODE_ OFF_PATTERN	MSRASTMODE_ ON_PIXEL	MSRASTMODE_ ON_PATTERN	*	*	*
WM_INT::WM_HZ_OP	*	*	*	*	True	True	False
3DSTATE_WM_HZ_OP:: Number of Multisamples	*	*	*	*	> NUMSAMPLES_1	NUMSAMPLES_1	*
WM_INT::Multisample Rasterization Mode	OFF_PIXEL	OFF_PATTERN	ON_PIXEL	ON_PATTERN	ON_PATTERN	ON_PIXEL	Determined from Table 1 in 3D Pipeline Windower > Multisampling > Multisample Modes/State)

Note: OFF_PIXEL, OFF_PATTERN, ON_PIXEL, ON_PATTERN modes are described in 3D Pipeline Windower > Multisampling > Multisample Modes/State.

Signal	WM_INT::Multisample Dispatch Mode
Description	This bit, determines how PS threads are dispatched
Formula	= (WM_INT::RT Independent Rasterization Enable)? PerPixel: (3DSTATE_PS_EXTRA::PixelShaderIsPerSample) ? PerSample : PerPixel

Signal	WM_INT::Pixel Shader Kill Pixel
Description	This bit, if ENABLED, indicates that the PS kernel or color calculator has the ability to kill (discard) pixels or samples, *other than due to depth or stencil testing*.
Formula	= (3DSTATE_WM::ForceKillPixel == ON) || ((3DSTATE_WM::ForceKillPixel != Off) && ! WM_INT::WM_HZ_OP && ! 3DSTATE_WM::EDSC_Mode == 2 && (WM_INT::Depth Write Enable || WM_INT::Stencil Write Enable) && (3DSTATE_PS_EXTRA::PixelShaderKillsPixels || 3DSTATE_PS_EXTRA:: **oMask Present to RenderTarget** || 3DSTATE_PS_BLEND::AlphaToCoverageEnable || 3DSTATE_PS_BLEND::AlphaTestEnable || 3DSTATE_WM_CHROMAKEY::ChromaKeyKillEnable))

Signal	WM_INT::Early Depth/Stencil Control
Description	This field specifies the behavior of early depth/stencil test.
Formula	= (WM_INT::WM_HZ_OP) ? EDSC_NORMAL : WM_INT::RT Independent Rasterization Enable ? EDSC_PSEXEC : 3DSTATE_WM::Early Depth/Stencil Control

Signal	WM_INT::RT Independent Rasterization Enable
Description	Enables Render Target Independent Rasterization.
Formula	= (WM_INT::WM_HZ_OP ? Disable : (3DSTATE_RASTER::ForceSampleCount != NUMRASTSAMPLES_0) ? Enable : Disable

Signal	WM_INT::Statistics Enable
Description	Enables Statistics
Formula	= (WM_INT::WM_HZ_OP) ? Disable : 3DSTATE_WM:: Statistics Enable

Signal	WM_INT::Polygon Stipple Enable
Description	Enables Poly Stipple
Formula	= (WM_INT::WM_HZ_OP) ? Disable : 3DSTATE_WM::Polygon Stipple Enable

Signal	WM_INT::WM_HZ_OP
Description	Enables WM_HZ_OP
Formula	= (3DSTATE_WM_HZ_OP::DepthBufferClear || 3DSTATE_WM_HZ_OP::DepthBufferResolve ||

	3DSTATE_WM_HZ_OP::Hierarchical Depth Buffer Resolve Enable || 3DSTATE_WM_HZ_OP::StencilBufferResolve || 3DSTATE_WM_HZ_OP::StencilBufferClear || 3DSTATE_WM_HZ_OP::DepthBufferPartialResolve) ? Enable : Disable

Signal	WM_INT:: Pixel Location
Description	Sets the input pixel location to Center if UL and doing multisampling
Formula	(3DSTATE_MULTISAMPLE::Pixel Location && 3DSTATE_MULTISAMPLE::Pixel Position Offset Enable && WM_MULTISAMPLE_INT::Number of Multisamples > 0) ? 0 : 3DSTATE_MULTISAMPLE::Pixel Location

3DSTATE_SAMPLE_MASK

The following inline state packets define the sample mask state used by the windower stage for different generations.

State Packets
3DSTATE_SAMPLE_MASK

Signal	WM_INT:: Sample Mask Enable
Description	Sets Sample Mask used in rasterization

Formula	Switch(WM_MULTISAMPLE_INT::Number of Multisamples
	{
	Case NUMSAMPLES_1: WM_INT:: Sample Mask Enable = 0x0001; break;
	Case NUMSAMPLES_2: WM_INT:: Sample Mask Enable = 0x0003; break;
	Case NUMSAMPLES_4: WM_INT:: Sample Mask Enable = 0x000F; break;
	Case NUMSAMPLES_8: WM_INT:: Sample Mask Enable = 0x00FF; break;
	Add this additional case:
	Case NUMSAMPLES_16: WM_INT:: Sample Mask Enable = 0xFFFF; break;
	}

Signal	WM_INT:: Sample Mask
Description	Sets Sample Mask used in rasterization
Formula	= WM_INT:: Sample Mask Enable & (WM_INT::WM_HZ_OP) ? 3DSTATE_WM_HZ_OP::Sample Mask: 3DSTATE_SAMPLE_MASK::Sample Mask)

3DSTATE_WM_CHROMAKEY

3DSTATE_WM_HZ_OP

State Restrictions

State	Restriction
3DSTATE_PS::Render Target Fast Clear Enable	Must be disabled
3DSTATE_PS:: Render Target Resolve Enable	Must be disabled
3DSTATE_WM:: Legacy Depth Buffer Clear	Must be disabled
3DSTATE_WM:: Legacy Depth Buffer Resolve	Must be disabled
3DSTATE_WM:: Legacy Hierarchical Depth Buffer Resolve Enable	Must be disabled
3DSTATE_MULTISAMPLE::Pixel Location	Must be set according to the API being used.

State Overrides

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable (full or partial)	Hierarchical Depth Buffer Resolve Enable
SF_INT:: Statistics Enable	Disable	Disable	Disable	Disable
SF_INT:: View Transform Enable	Disable	Disable	Disable	Disable
SF_INT::Multisample Rasterization Mode	(3DSTATE_WM_HZ_OP::NumberOfSamples > 1) ? ON_PATTERN : ON_PIXEL	(3DSTATE_WM_HZ_OP::NumberOfSamples > 1) ? ON_PATTERN : ON_PIXEL	(3DSTATE_WM_HZ_OP::NumberOfSamples > 1) ? ON_PATTERN : ON_PIXEL	(3DSTATE_WM_HZ_OP::NumberOfSamples > 1) ? ON_PATTERN : ON_PIXEL
SF_INT::Cull Mode	CULLMODE_BACK	CULLMODE_BACK	CULLMODE_BACK	CULLMODE_BACK
SF_INT::Scissor Rectangle Enable	3DSTATE_WM_HZ_OP:: Scissor Rectangle Enable	3DSTATE_WM_HZ_OP:: Scissor Rectangle Enable	3DSTATE_WM_HZ_OP:: Scissor Rectangle Enable	3DSTATE_WM_HZ_OP:: Scissor Rectangle Enable
SF_INT::RT Independent Rasterization Enable	Disable	Disable	Disable	Disable
SF_INT::FrontFace Fill Mode	SOLID	SOLID	SOLID	SOLID

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable (full or partial)	Hierarchical Depth Buffer Resolve Enable
SF_INT::FrontWinding	FRONTWINDING_CW	FRONTWINDING_CW	FRONTWINDING_CW	FRONTWINDING_CW
SF_INT::Render Target Array index	0	0	0	0
SF_INT::Viewport index	0	0	0	0
SF_INT:: Geometry Hashing Disable	Disable	Disable	Disable	Disable
WM_INT::StencilTest Enable	Enable	Stencil buffer Clear ? Enable : Disable	Disable	Disable
WM_INT::StencilWriteEnable	Enable	Stencil buffer Clear ? Enable : Disable	Disable	Disable
WM_INT::DepthTest Enable	Disable	Disable	Enable	Disable
WM_INT::DepthWriteEnable	Depth buffer Clear ? Enable : Disable	Enable	Enable	Enable
WM_INT::DepthTest Function	NEVER	NEVER	NEVER	NEVER
WM_INT::StencilTest Function	ALWAYS	Stencil buffer Clear ? ALWAYS: No Override	No Override	No Override
WM_INT::StencilPassDepthPassOp	REPLACE	Stencil buffer Clear ? REPLACE: No Override	No Override	No Override
WM_INT:: Statistics Enable	Disable	Disable	Disable	Disable
WM_INT::ThreadDispatchEnable	Disable	Disable	Disable	Disable
WM_INT:: Pixel Shader Kill Pixel	Disable	Disable	Disable	Disable
WM_INT:: Pixel Shader Computed Depth Mode	PSCDEPTH_OFF	PSCDEPTH_OFF	PSCDEPTH_OFF	PSCDEPTH_OFF
WM_INT: Computed	Disable	Disable	Disable	Disable

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable (full or partial)	Hierarchical Depth Buffer Resolve Enable
Stencil Enable				
WM_INT::RT Independent Rasterization Enable	Disable	Disable	Disable	Disable
WM_INT::Polygon Stipple Enable	Disable	Disable	Disable	Disable
WM_INT::Multisample Rasterization Mode	NumberOfSamples > 0 ? ON_PATTERN : ON_PIXEL)	NumberOfSamples > 0 ? ON_PATTERN : ON_PIXEL)	NumberOfSamples > 0 ? ON_PATTERN : ON_PIXEL)	NumberOfSamples > 0 ? ON_PATTERN : ON_PIXEL)
MULTISAMPLE_INT:: Number of Multisamples	3DSTATE_WM_HZ_OP:: Number of Multisamples	3DSTATE_WM_HZ_OP:: Number of Multisamples	3DSTATE_WM_HZ_OP:: Number of Multisamples	3DSTATE_WM_HZ_OP:: Number of Multisamples
WM_INT::Sample Mask	3DSTATE_WM_HZ_OP::Sample Mask	3DSTATE_WM_HZ_OP::Sample Mask	3DSTATE_WM_HZ_OP::Sample Mask	3DSTATE_WM_HZ_OP::Sample Mask
WM_INT::Early Depth/Stencil Control	EDSC_NORMAL	EDSC_NORMAL	EDSC_NORMAL	EDSC_NORMAL
WM_INT:: Full Surface Depth Clear	Depth buffer clear ? WM_HZ_OP:: Full Surface Depth Clear : Disable	WM_HZ_OP:: Full Surface Depth Clear	Disable	Disable
WM_INT:: Full Surface Depth Clear	Depth buffer clear ? WM_HZ_OP:: Full Surface Depth Clear : Disable	WM_HZ_OP:: Full Surface Depth Clear	Disable	Disable

3DSTATE_WM_DEPTH_STENCIL

Signal	WM_INT::DepthTestEnable
Description	Enables Depth Test
Formula	= (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE != NULL) && (WM_INT::WM_HZ_OP ? Use the WM_INT::DepthTestEnable from WM_HZ_OP table : ((3DSTATE_WM_DEPTH_STENCIL::DepthTestEnable || WM_INT::DBT_ENABLE) && ! WM_INT::RT Independent Rasterization Enable))

Signal	WM_INT::DepthTestFunction
Description	Depth Test Function
Formula	WM_INT::WM_HZ_OP ? Use the WM_INT::DepthTestFunction from WM_HZ_OP table : (((!3DSTATE_WM_DEPTH_STENCIL::DepthTestEnable && WM_INT::DBT_enable) ? 3D_Compare_Function:: COMPAREFUNCTION_ALWAYS : 3DSTATE_WM_DEPTH_STENCIL::DepthTestFunction)

Signal	WM_INT::DBT_enable
Description	Depth Bounds Test enable
Formula	(WM_INT::WM_HZ_OP) ? 0 : 3DSTATE_DEPTH_BOUNDS::DBT_enable && (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE != NULL)

Signal	WM_INT::StencilWriteEnable
Description	Enables writes to the Stencil Buffer
Formula	= 3DSTATE_STENCIL_BUFFER::SURFACE_TYPE != NULL && 3DSTATE_STENCIL_BUFFER::STENCIL_WRITE_ENABLE && ((WM_INT::WM_HZ_OP ? Use the WM_INT::StencilWriteEnable from WM_HZ_OP table : WM_INT::StencilTestEnable && 3DSTATE_WM_DEPTH_STENCIL::StencilBufferWriteEnable))

Signal	WM_INT::StencilWriteEnable
Description	Enables writes to the Stencil Buffer
Formula	= 3DSTATE_STENCIL_BUFFER::SURFACE_TYPE != NULL && 3DSTATE_STENCIL_BUFFER::STENCIL_WRITE_ENABLE && (

	(WM_INT::WM_HZ_OP ? Use the WM_INT::StencilWriteEnable from WM_HZ_OP table : WM_INT::StencilTestEnable && 3DSTATE_WM_DEPTH_STENCIL::StencilBufferWriteEnable))

Signal	WM_INT::DepthWriteEnable
Description	Enables Depth Write
Formula	= (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE != NULL) && 3DSTATE_DEPTH_BUFFER::DEPTH_WRITE_ENABLE && (WM_INT::WM_HZ_OP ? Use the WM_INT::DepthWriteEnable from WM_HZ_OP table : 3DSTATE_WM_DEPTH_STENCIL::DepthWriteEnable)

Signal	WM_INT::StencilTestFunction
Description	Stencil Test Function
Formula	WM_INT::WM_HZ_OP ? Use the WM_INT::StencilTestFunction from WM_HZ_OP table : 3DSTATE_WM_DEPTH_STENCIL::StencilTestFunction

Signal	WM_INT::StencilPassDepthPassOp
Description	StencilPassDepthPassOp
Formula	WM_INT::WM_HZ_OP ? Use the WM_INT::StencilPassDepthPassOp from WM_HZ_OP table : 3DSTATE_WM_DEPTH_STENCIL::StencilPassDepthPassOp

Signal	WM_INT::Stencil Test Mask
Description	Stencil test Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL::Stencil Test Mask

Signal	WM_INT::Stencil Write Mask
Description	Stencil Write Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL::Stencil Write Mask

Signal	WM_INT::BackFace Stencil Test Mask
Description	Stencil test Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL:: Backface Stencil Test Mask

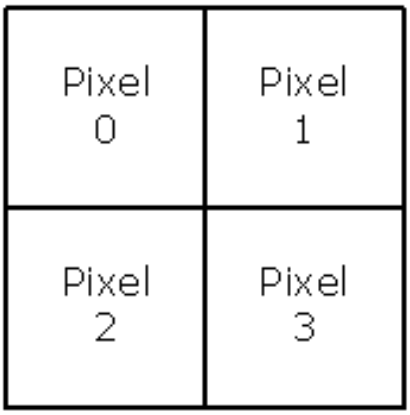
Signal	WM_INT:: BackFace Stencil Write Mask
Description	Stencil Write Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL::Backface Stencil Write Mask

Rasterization

The WM unit uses the setup computations performed by the SF unit to rasterize objects into the corresponding set of pixels Most of the controls regarding the screen-space geometry of rendered objects are programmed via the SF unit.

The rasterization process generates pixels in 2x2 groups of pixels called *subspans* (see *Pixels with a SubSpan* below) which, after being subjected to various inclusion/discard tests, are grouped and passed to spawned Pixel Shader (PS) threads for subsequent processing Once these PS threads are spawned, the WM unit provides only bookkeeping functions on the pixels Note that the WM unit can proceed on to rasterize subsequent objects while PS threads from previous objects are still executing.

Pixels with a SubSpan



B6850-01

Drawing Rectangle Clipping

The Drawing Rectangle defines the maximum extent of pixels which can be rendered Portions of objects falling outside the Drawing Rectangle will be clipped (pixels discarded) Implementations will typically discard objects falling completely outside of the Drawing Rectangle as early in the pipeline as possible There is no control to turn off Drawing Rectangle clipping - it is unconditional.

For the purposes of clipping, the Drawing Rectangle must itself be clipped to the destination buffer extents (The Drawing Rectangle Origin, used to offset relative X,Y coordinates earlier in the pipeline, is permitted to lie offscreen). The **Clipped Drawing Rectangle X,Y Min,Max** state variables (programmed via 3DSTATE_DRAWING_RECTANGLE - See *SF Unit*) defines the intersection of the Drawing Rectangle and the Color Buffer It is specified with non-negative integer pixel coordinates relative to the Destination Buffer upper-left origin.

Pixels with coordinates outside of the Drawing Rectangle cannot be rendered (i.e., the rectangle is inclusive) For example, to render to a full-screen 1280x1024 buffer, the following values would be required: Xmin=0, Ymin=0, Xmax=1279 and Ymax=1023

For "full screen" rendering, the Drawing Rectangle coincides with the screen-sized buffer For "front-buffer windowed" rendering it coincides with the destination "window".

Line Rasterization

See *SF Unit* chapter for details on the screen-space geometry of the various line types.

Coverage Values for Anti-Aliased Lines

The WM unit is provided with both the **Line Anti-Aliasing Region Width** and **Line End Cap Anti-aliasing Region Width** state variables (in WM_STATE) in order to compute the coverage values for anti-aliased lines.

3DSTATE_AA_LINE_PARAMS

3DSTATE_AA_LINE_PARAMETERS

The slope and bias values should be computed to closely match the reference rasterizer results Based on empirical data, the following recommendations are offered:

The final alpha for the center of the line needs to be 148 to match the reference rasterizer In this case, the Lo to edge 0 and edge 3 will be the same Since the alpha for each edge is multiplied together, we get:

edge0alpha * edge1alpha = 148/255 = 0.580392157

Since edge0alpha = edge3alpha we get:

$(edge0alpha)^2$ = 0.580392157

edge0alpha = sqrt(0.580392157) = 0.761834731 at the center pixel

The desired alpha for pixel 1 = 54/255 = 0.211764706

The slope is (0.761834731 - 0.211764706) = 0.550070025

Since we are using 8 bit precision, the slope becomes

AA Coverage [EndCap] Slope = 0.55078125

The alpha value for Lo = 0 (second pixel from center) determines the bias term and is equal to

(0.211764706 - 0.550070025) = -0.338305319

With 8 bits of precision the programmed bias value

Line Stipple

Line stipple, controlled via the **Line Stipple Enable** state variable in WM_STATE, discards certain pixels that are produced by non-AA line rasterization.

The line stipple rule is specified via the following state variables programmed via 3DSTATE_LINE_STIPPLE: the 16-bit **Line Stipple Pattern** (p), **Line Stipple Repeat Count** l, and **Line Stipple Inverse Repeat Count.** Software must compute **Line Stipple Inverse Repeat Count** as 1.0f / **Line Stipple Repeat Count** and then converted from float to the required fixed-point encoding (see 3STATE_LINE_STIPPLE).

The WM unit maintains an internal Line Stipple Counter state variable (s) The initial value of s is zero; s is incremented after production of each pixel of a line segment (pixels are produced in order, beginning at the starting point and working towards the ending point). S is reset to 0 whenever a new primitive is processed (unless the primitive type is LINESTRIP_CONT or LINESTRIP_CONT_BF), and before every line segment in a group of independent segments (LINELIST primitive).

During the rasterization of lines, the WM unit computes:

$$b = \lfloor s/r \rfloor \bmod 16,$$

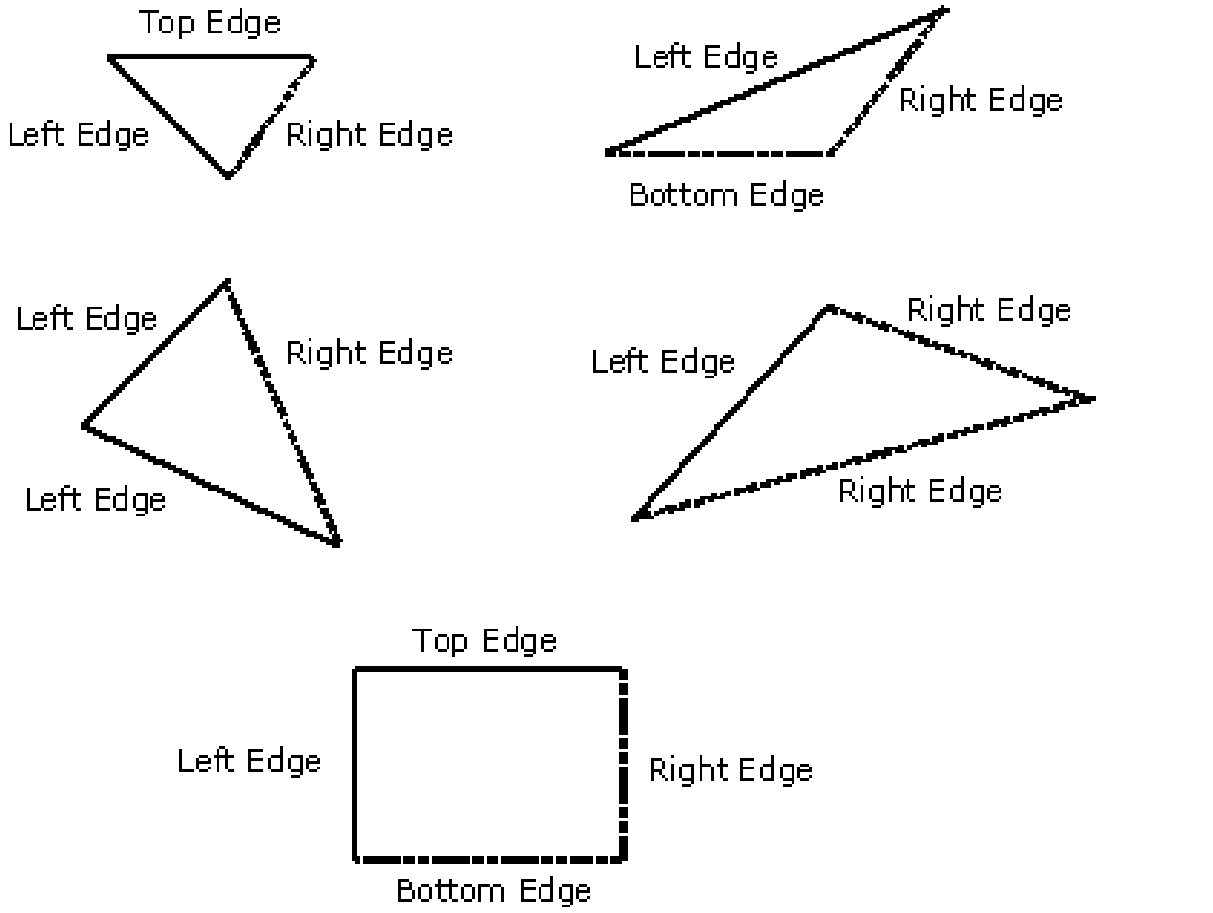
A pixel is rendered if the bth bit of p is 1, otherwise it is discarded. The bits of p are numbered with 0 being the least significant and 15 being the most significant.

Polygon (Triangle and Rectangle) Rasterization

The rasterization of LINE, TRIANGLE, and RECTANGLE objects into pixels requires a "pixel sampling grid" to be defined This grid is defined as an axis-aligned array of pixel sample points spaced exactly 1 pixel unit apart If a sample point falls within one of these objects, the pixel associated with the sample point is considered "inside" the object, and information for that pixel is generated and passed down the pipeline

For TRIANGLE and RECTANGLE objects, if a sample point intersects an edge of the object, the associated pixel is considered "inside" the object if the intersecting edge is a "left" or "top" edge (or, more exactly, the intersected edge is not a "right" or "bottom" edge) Note that "top" and "bottom" edges are by definition exactly horizontal. See *TRIANGLE and RECTANGLE Edge Types* below for the edge types for representative TRIANGLE and RECTANGLE objects (solid edges are inclusive, dashed edges are exclusive).

TRIANGLE and RECTANGLE Edge Types



Polygon Stipple

The *Polygon Stipple* function, controlled via the **Polygon Stipple Enable** state variable in WM_STATE, allows only selected pixels of a repeated 32x32 pixel pattern to be rendered Polygon stipple is applied only to the following primitive types:

3DPRIM_POLYGON
3DPRIM_TRIFAN
3DPRIM_TRILIST
3DPRIM_TRISTRIP
3DPRIM_TRISTRIP_REVERSE

Note that the 3DPRIM_TRIFAN_NOSTIPPLE object is never subject to polygon stipple.

The stipple pattern is defined as a 32x32 bit pixel mask via the 3DSTATE_POLY_STIPPLE_PATTERN command. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

The origin of the pattern is specified via **Polygon Stipple X,Y Offset** state variables programmed via the 3DSTATE_POLY_STIPPLE_OFFSET command The offsets are pixel offsets from the Color Buffer origin to the upper left corner of the stipple pattern. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

3DSTATE_POLY_STIPPLE_OFFSET

3DSTATE_POLY_STIPPLE_PATTERN

Multisampling

The multisampling function has two components:

- **Multisample Rasterization**: multisample rasterization occurs at a subpixel level, wherein each pixel consists of a number of "samples" at state-defined positions within the pixel footprint. Coverage of the primitive as well as color calculator operations (stencil test, depth test, color buffer blending, etc.) are done at the sample level. In addition the pixel shader itself can optionally run at the sample level depending on a separate state field.
- **Multisample Render Targets (MSRT)**: The render targets, as well as the depth and stencil buffers, now have the ability to store per-sample values. When combined with multisample rasterization, color calculator operations such as stencil test, depth test, and color buffer blending are done with the destination surface containing potentially different values per sample.

3DSTATE_MULTISAMPLE

Signal	WM_MULTISAMPLE_INT::Number of Multisamples
Description	Set the number of multisamples
Formula	= (WM_INT::WM_HZ_OP) ? 3DSTATE_ WM_HZ_OP::Number of Multisamples : 3DSTATE_MULTISAMPLE::Number of Multisamples

3DSTATE_SAMPLE_PATTERN

Multisample ModesState

A number of state variables control the operation of the multisampling function. The following table indicates the states and their location. Refer to the state definition for more details.

State Element	Source	Description
Multisample Rasterization Mode	WM_INT::Multisample Rasterization Mode	Controls whether rasterization of non-lines is performed on a pixel or sample basis (PIXEL vs. PATTERN), and whether multisample rasterization of lines is enabled (OFF vs. ON). From this generation forward, this state element becomes an internal signal computed by other state variables (also listed here) unless certain modes are set, which can be seen in the WM_INT equation for the signal.
Multisample Dispatch Mode	WM_INT::Multisample Dispatch Mode	Controls whether the pixel shader is executed per pixel or per sample.
Number of Multisamples	3DSTATE_MULTISAMPLE and SURFACE_STATE	Indicates the number of samples per pixel contained on the surface. This field in 3DSTATE_MULTISAMPLE must match the corresponding field in SURFACE_STATE for each render target. The depth, hierarchical depth, and stencil buffers inherit this field from 3DSTATE_MULTISAMPLE.
RTIR Enabled	3DSTATE_RASTER::ForcedSampleCount != NUMRASTSAMPLES_0	Enable Render Target Independent Rasterization.
Pixel Location	3DSTATE_MULTISAMPLE	Indicates the subpixel location where values specified as "pixel" are sampled. This is either the upper left corner or the center.
MSAA Sample Offsets	3DSTATE_SAMPLE_PATTERN	For each of the N samples, specifies the subpixel location of each sample.
RTIR Sample Offsets	3DSTATE_SAMPLE_PATTERN	For each of the N samples, specifies the subpixel location of each sample.
API Mode	3DSTATE_RASTER	One of the deciding factors of what the Multisample Rasterization Mode should be according to WM_INT::Multisample Rasterization Mode. Software sets this field according to the API's version.

State Element	Source	Description
DX Multisample Rasterization Enable	3DSTATE_RASTER	Controls ON/OFF part of Multisample Rasterization Mode, depending on the API Mode according to WM_INT::Multisample Rasterization Mode.

This table does not apply if (3DSTATE_RASTER::ForceMultisampleRasterMode == Force) or (WM_INT::WM_HZ_OP == true).

Table 1: Multisample Rasterization Modes

Number of Multisamples	NUMSAMPLES_1	NUMSAMPLES_1	> NUMSAMPLES_1	> NUMSAMPLES_1	Any	Any	Any	Any
DX Multisample Rasterization Enable	0	1	0	1	0	1	0	1
Rast Number of Samples	Disabled	Disabled	Disabled	Disabled	NUMRAST SAMPLES_1	NUMRAST SAMPLES_1	> NUMRAST SAMPLES_1	> NUMRAST SAMPLES_1
API Mode == DX9.0/OGL	OFF_PIXEL	OFF_PIXEL	OFF_PIXEL	ON_PATTERN	Invalid	Invalid	Invalid	Invalid
API Mode == DX10.0	OFF_PIXEL	ON_PIXEL	OFF_PIXEL	ON_PATTERN	OFF_PIXEL	Invalid	Invalid	Invalid
API Mode == DX10.1+/Vulkan	OFF_PIXEL	ON_PIXEL	OFF_PATTERN	ON_PATTERN	OFF_PIXEL	ON_PIXEL	OFF_PATTERN	ON_PATTERN

Definitions for lines terms used in Table 2 through Table 4:

- **Legacy Lines:** Way of drawing lines that allows Diamond Lines (SF_STATE::Line Width == 0.0), Non-anti-aliased Wide Lines (SF_STATE::Line Width != 0.0), and Line Stippling (3DSTATE_WM:: Line Stipple Enable == 1).
- **AA Lines:** Way of drawing lines that allows Anti-aliased line. These are lines rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices with coverage value (referred to as Anti-alias Alpha) computed per pixel.

AA Line Support Requirement
SF_INT::Anti-aliasing Enable == 1

- **MSAA Lines:** Way of drawing lines that allows Multisample Anti-aliased lines. These are lines rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices, but no Anti alias alpha coverage is computed.

Table 2: Type of Line Algorithm Given an Arrangement of State Variables

Multisample Rasterization Mode	Anti-Aliasing Enable	SF_STATE::Line Width	Line Algorithm
OFF_*	0	Non-Zero	Non-Anti-aliased Wide Lines
OFF_*	0	0.0	Diamond Lines
OFF_*	1	Non-Zero	See Note A below.
OFF_*	1	0.0	Diamond Lines
ON_*	*	*	MSAA Lines

Note A: Anti-Aliasing Details for Table 2

Anti-Aliasing Details
Anti-Aliased Lines with Alpha Coverage

Table 3: Multisample Modes with RTIR Disabled

Number of Multisamples	MS RAST MODE	MS DISP MODE	HW Mode
NUMSAMPLES_1	OFF_PIXEL	PERSAMPLE	**Legacy Non-MSAA Mode** 1X rasterization, using Pixel Location Legacy lines or AA-line rasterization 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	ON_PIXEL	PERSAMPLE	**1X Multisampling Mode** 1X rasterization, using Pixel Location MSAA lines only, using Pixel Location 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	-	PERPIXEL	Treated the same as PERSAMPLE
	ON_PATTERN	-	Invalid
	OFF_PATTERN	-	Invalid
n where n > NUMSAMPLES_1	OFF_PIXEL	PERPIXEL	**MSRT Only, PerPixel PS** 1X rasterization, using Pixel Location See Note B below. 1X PS, sample at Pixel Location 4X output merge, eval Depth at Pixel Location

Number of Multisamples	MS RAST MODE	MS DISP MODE	HW Mode
		PERSAMPLE	**MSRT Only, PerSample PS** 1X rasterization, using Pixel Location See Note B below. nX PS, all samples at Pixel Location nX output merge, eval Depth at Pixel Location
	ON_PIXEL	PERPIXEL	**Multibuffering MSAA, PerPixel PS** 1X rasterization, using Pixel Location MSAA lines only 1X PS, sample at Pixel Location 4X output merge, eval Depth at Pixel Location
		PERSAMPLE	**Multibuffering MSAA, PerSample PS** 1X rasterization, using Pixel Location MSAA lines only nX PS, all samples at Pixel Location nX output merge, eval Depth at Pixel Location
	OFF_PATTERN	PERPIXEL	**Mixed Mode, PerPixel PS** See Note B below. Non-Lines: nX rasterization, using Sample Offsets 1X PS, sample at Pixel Location nX output merge, eval depth at Sample Offsets
		PERSAMPLE	**Mixed Mode, PerSample PS** See Note B below. Non-Lines: nX rasterization, using Sample Offsets nX PS, sample at Pixel Location or Sample Offsets nX output merge, eval depth at Sample Offsets
	ON_PATTERN	PERPIXEL	**Pattern MSAA, PerPixel PS** nX rasterization, using Sample Offsets MSAA lines only 1X PS, sample at Pixel Location

Number of Multisamples	MS RAST MODE	MS DISP MODE	HW Mode
			nX output merge, eval depth at Sample Offsets
		PERSAMPLE	**Pattern MSAA, PerSample PS** nX rasterization, using Sample Offsets MSAA lines only nX PS, sample at Pixel Location or Sample Offsets nX output merge, eval depth at Sample Offsets

Note B: Line Details for Table 3 and Table 4

Line Details
Legacy lines or AA-line rasterization. For PERPIXEL or PERSAMPLE in Table 3 use pixel location. For OFF_PATTERN in Table 4 use pixel location.

Table 4: Multisample Modes with RTIR Enabled

Rast Number of Samples	MS RAST MODE	HW Mode
NUMRASTSAMPLES_1	OFF_PIXEL	**Legacy Non-MSAA Mode** 1X rasterization, using Pixel Location Legacy lines or AA-line rasterization 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	ON_PIXEL	**1X Multisampling Mode** 1X rasterization, using Pixel Location MSAA lines only, using Pixel Location 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	ON_PATTERN	Invalid
	OFF_PATTERN	Invalid
n where n > NUMRASTSAMPLES_1	OFF_PIXEL	**Invalid**
	ON_PIXEL	**Invalid**
	OFF_PATTERN	**Mixed Mode, PerPixel PS** See Note B above. Non-Lines: nX rasterization, using Sample Offsets

Rast Number of Samples	MS RAST MODE	HW Mode
		1X PS, sample at Pixel Location 1X output merge, eval depth atPixel Location
	ON_PATTERN	**Pattern RTIR, PerPixel PS** nX rasterization, using Sample Offsets MSAA lines only 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location

Note: Multisample Dispatch Mode is not taken into account in Table 4 given that with RTIR:

Details
The value of PERSAMPLE is converted to PERPIXEL internally.

Other WM Functions

The only other WM function is Statistics Gathering.

Statistics Gathering

If **Statistics Enable** is set in WM_STATE or 3DSTATE_WM, the Windower increments the PS_INVOCATIONS_COUNT register once for each unmasked pixel (or sample) that is *dispatched* to a Pixel Shader thread.

If **Early Depth Test Enable** is set it is possible for pixels or samples to be discarded before reaching the Pixel Shader due to failing the depth or stencil test. PS_INVOCATIONS_COUNT will still be incremented for these pixels or samples since the depth test occurs after the pixel shader from the point of view of SW.

Pixel

This section contains the following subsections:

- **Depth and Stencil**, which covers the Depth and Stencil test functions
- **Pixel Dispatch**, which covers pixel shader state, pixel grouping, multisampling effects on pixel shader dispatch, and pixel shader thread payload
- **Pixel Backend**, which covers backend processing

Pixel Hashing

A block of pixel is hashed to slice and subslice based on screenspace X,Y. **Cross Slice Hashing Mode** and **Subslice Hashing Mode** indicate the size and shape of the pixel block used for the hash. Subslice

hashing is calculated independently from slice hashing, but together determine how pixel workload is distributed to all the subslices.

3DSTATE_3D_MODE

X,Y shown below is the pixel block address.

SubSlice Hashing

2-way Hashing

subslice_id = X[0] ^ Y[0]

3-way Hashing

If X+Y is divisible by 3, then subslice_id=0, else subslice_id = 1 + X[2]^Y[0]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	2	2	0	2	1	0	1	1	0	2	2	0
1	2	2	0	2	1	0	1	1	0	2	2	0	1	1	0	1
2	1	0	1	1	0	2	2	0	1	1	0	1	2	0	2	2
3	0	2	2	0	1	1	0	1	2	0	2	2	0	1	1	0
4	1	1	0	1	2	0	2	2	0	1	1	0	2	2	0	2
5	2	0	2	2	0	1	1	0	2	2	0	2	1	0	1	1
6	0	1	1	0	2	2	0	2	1	0	1	1	0	2	2	0
7	2	2	0	2	1	0	1	1	0	2	2	0	1	1	0	1
8	1	0	1	1	0	2	2	0	1	1	0	1	2	0	2	2
9	0	2	2	0	1	1	0	1	2	0	2	2	0	1	1	0
10	1	1	0	1	2	0	2	2	0	1	1	0	2	2	0	2
11	2	0	2	2	0	1	1	0	2	2	0	2	1	0	1	1
12	0	1	1	0	2	2	0	2	1	0	1	1	0	2	2	0
13	2	2	0	2	1	0	1	1	0	2	2	0	1	1	0	1
14	1	0	1	1	0	2	2	0	1	1	0	1	2	0	2	2
15	0	2	2	0	1	1	0	1	2	0	2	2	0	1	1	0

Programmable DualSubSlice Hashing

In addition to the above calculated dualsubslice_id, when enabled via **Subslice Hashing Table Enable**, dualsubslice_id is indicated by the entry in the dualsubslice_id hashing tables. X,Y address used to index the table is either 8x8 or 16x16 pixel block.

DUALSUBSLICE_HASH_TABLE_8x8

DUALSUBSLICE_HASH_TABLE_16x8

Slice Hashing

Table based Hashing

Slice_id hash is a lookup into a 256 entry slice_hash_table. The lowest 4 bits of the pixel block X,Y is used to index into the 16x16 table.

If (16x16 hashing)

X[3:0] = x[7:4]

Y[3:0] = y[7:4]

Else if (32x32 hashing)

X[3:0] = x[8:5]

Y[3:0] = y[8:5]

endif

Slice = Slice Hash Table[Y][X] / NUM_HZ_IN_A_SLICE

Hz Slice = Slice Hash Table[Y][X] % NUM_HZ_IN_A_SLICE

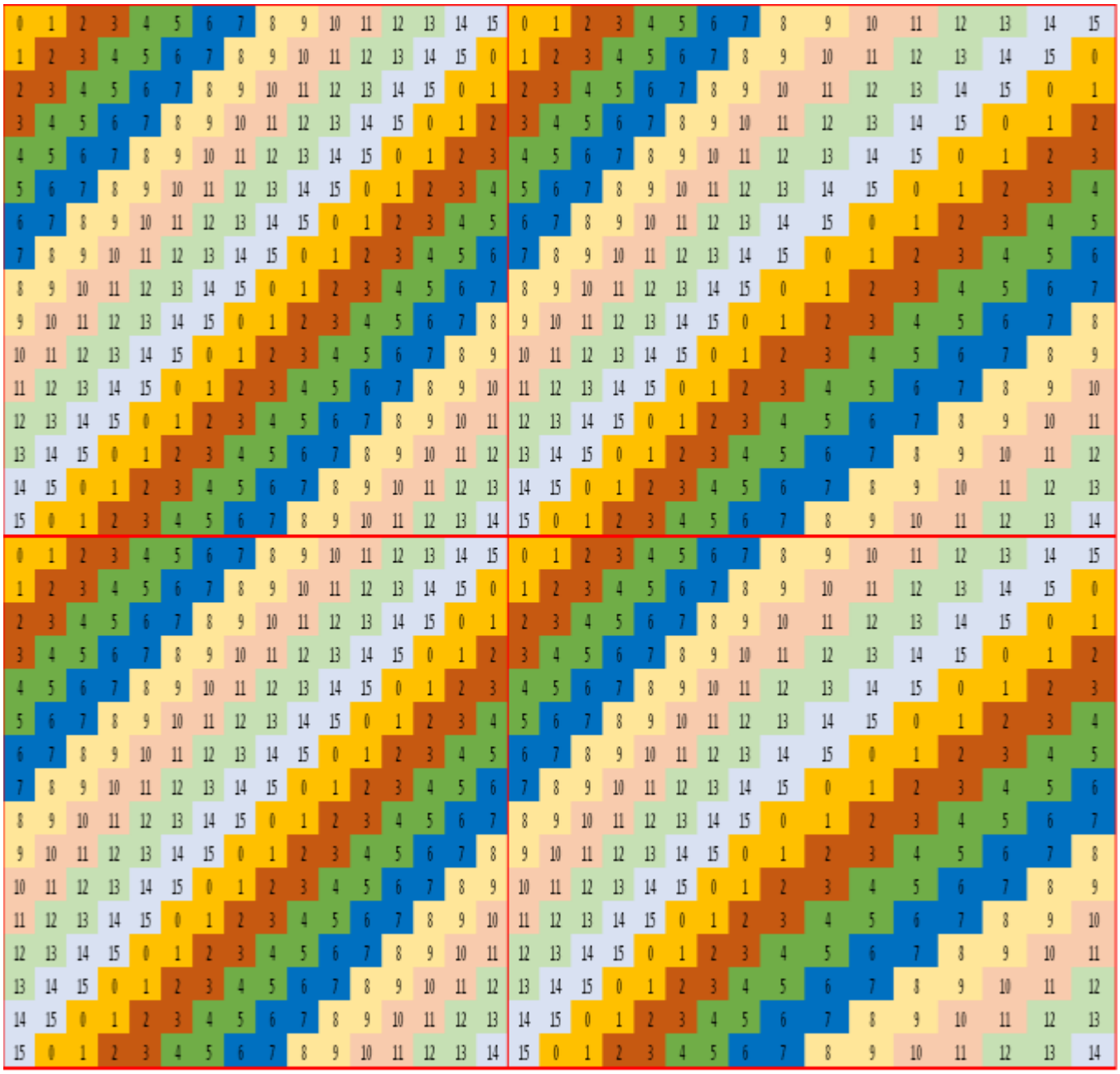
Default ROM Table

When **Slice Hashing Table Enable** is set to false, slice_hash_table defaults to ROM tables based on the current number of Z Pipe Id, and the following effective slice_id.

Current default hash table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
3	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
4	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
5	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
6	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
7	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6
8	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
9	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8
10	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
11	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10
12	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
13	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
14	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
15	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If we replicate the table:



In case of shutdown Hz Pipes.

A Hz Pipe is shutdown if both DSS are disabled.

The following algorithm defines a default mode, but it splits the render target into all enabled Hz Pipes without considering the number of DSS/Hz.

Only the Fuse Configuration is used for calculating Z-Pipe hashing and Dynamic Slice Shutdown has no effect.

If the list of enabled Hz pipes is pm_mode_dualsubsliceen

The following algorithm shows how to create the default table:

NUM_SUBSLICES = GT_NUM_SLICES*GT_SC_Z_PER_SLICE

Enabled_Z <-- 0

For i from 0 to 15

While (not pm_mode_sliceen[Enabled_Z/GT_NUM_GSLICES][Enabled_Z % GT_SC_Z_PER_SLICE])

Enabled_Z <-- (Enabled_Z + 1) % NUM_SUBSLICES

End while

Physical_array[i] <-- Enabled_Z

Enabled_Z <-- (Enabled_Z + 1) % NUM_SUBSLICES

End For

For y from 0 to 15

For x from 0 to 15

Slice Hash Table[Y][X] <-- Physical_array[(y+x) % NUM_SUBSLICES]

End For

End For

For Example:

In this example pm_mode_sliceen is: 0x4063

Physical_array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0

The table is:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0
1	1	5	6	14	0	1	5	6	14	0	1	5	6	14	0	0
2	5	6	14	0	1	5	6	14	0	1	5	6	14	0	0	1
3	6	14	0	1	5	6	14	0	1	5	6	14	0	0	1	5
4	14	0	1	5	6	14	0	1	5	6	14	0	0	1	5	6
5	0	1	5	6	14	0	1	5	6	14	0	0	1	5	6	14
6	1	5	6	14	0	1	5	6	14	0	0	1	5	6	14	0
7	5	6	14	0	1	5	6	14	0	0	1	5	6	14	0	1
8	6	14	0	1	5	6	14	0	0	1	5	6	14	0	1	5
9	14	0	1	5	6	14	0	0	1	5	6	14	0	1	5	6
10	0	1	5	6	14	0	0	1	5	6	14	0	1	5	6	14
11	1	5	6	14	0	0	1	5	6	14	0	1	5	6	14	0
12	5	6	14	0	0	1	5	6	14	0	1	5	6	14	0	1
13	6	14	0	0	1	5	6	14	0	1	5	6	14	0	1	5
14	14	0	0	1	5	6	14	0	1	5	6	14	0	1	5	6
15	0	0	1	5	6	14	0	1	5	6	14	0	1	5	6	14

This only a default function programming table, but for performance it needs to be optimized and reprogrammed based on the shutdown scenarios.

Programmable Hashing

Programmable Hashing

When **Slice Hashing Table Enable** is set to true, slice_hash_table is fetched via indirect state. An array of slice_hash_tables is stored at memory pointed to by **Slice Hash Table State Pointer**. All Slice Hashing Tables should have the same values. No slice_hash_table is fetched when there is only one active slice.

DSTATE_SLICE_TABLE_STATE_POINTERS

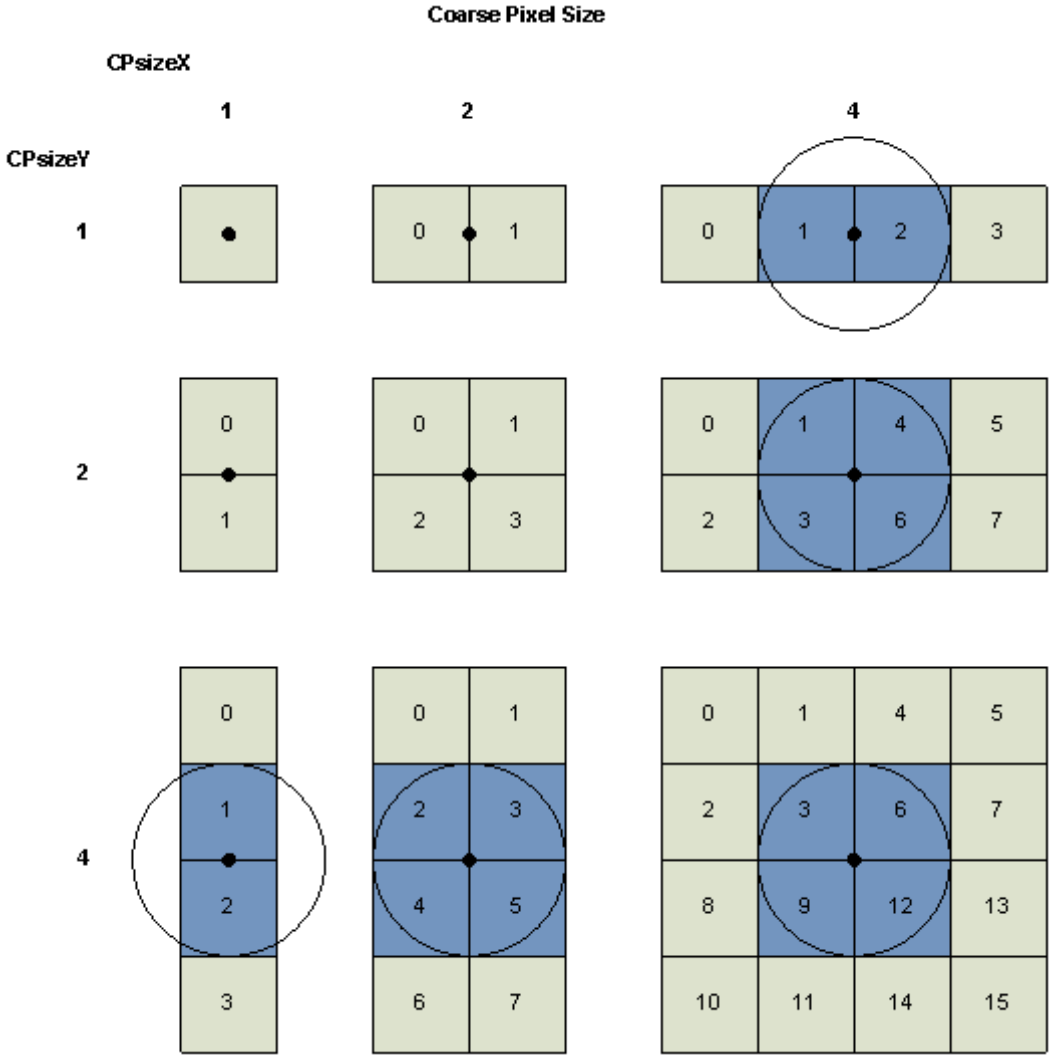
SLICE_HASH_TABLE - SLICE_HASH_TABLE

Coarse Pixel Shading

In Prior products, Pixel Shader can be invoked at either pixel frequency or at sample frequency. In general, finer grain shading creates more BW and compute demands in the graphics sub-system. In certain use cases, it is possible to do coarser grain shading than pixel without noticeable change to the image quality. This observation leads to HW support for coarser than pixel grain shading rate. It is called Coarse Pixel Shading (CPS).

Coarse Pixels

Similar to how pixels consist of multiple samples under MSAA, a coarse pixel consists of several pixels. Coarse pixel size is defined in terms of pixels by an ordered pair, e.g. (2,4) means CP has 2 pixels in X-axis and 4 pixels in Y-axis. Allowable CP sizes are from the set {1,2,4} X {1,2,4}. When CPS is enabled, HW computes CPsizes and gathers visible pixels to form CPs.



CPS Modes

Coarse Pixel Shading requires per primitive CP sizes to be determined. CPsize is fixed for a block of 8X8 aligned pixel block. There are two modes for determining the CPsize:

Constant: Entire RTV has the flat CPsizes as defined by the 3DSTATE_CPS. This mode can be used to under-shade uniformly.

Radial: Since camera focal plane tends to carry more detailed information in the rendered image, this mode provides increasing CPsizes as distance from the focal point (defined in 3DSTATE_PS) increases. Therefore shading rate decreases as pixels fall farther from the focal point. In this mode, CPsize is a function of the fragment's position.XY. Detailed computation with respect to parameters from 3DSTATE_CPS is described below:

First define the following variables:

- Let (Px, Py) be the (x,y) pixel position of the pixel for which the *requested coarse pixel size* is being computed, in pixel coordinates

- Let (Cx, Cy) be the center position, in pixel coordinates, specified as CoarsePixelSizeState.CenterX and CoarsePixelSizeState.CenterY. These values must be in [0.0f, MAX_RENDER_TARGET_LINEAR_RESOLUTION].
- Let (S_x^{min}, S_y^{min}) be the maximum coarse pixel size specified as (CoarsePixelSizeState.MinSizeX, CoarsePixelSizeState.MinSizeY). These values must be in [1.0f, 4.0f] (the minimum and maximum allowed coarse pixel sizes).
- Let (S_x^{max}, S_y^{max}) be the minimum coarse pixel size specified as (CoarsePixelSizeState.MaxSizeX, CoarsePixelSizeState.MaxSizeY). These values must be in [1.0f, 4.0f] (the minimum and maximum allowed coarse pixel sizes).
- Let R_{min} be the minimum radius specified as CoarsePixelSizeState.RadiusMinSize. These values must be in [0.0f, MAX_RENDER_TARGET_LINEAR_RESOLUTION * sqrt(2.0f)] (the minimum and maximum radii in pixel coordinates).
- Let R_{max} be the maximum radius specified as CoarsePixelSizeState.RadiusMaxSize. These values must be in [0.0f, MAX_RENDER_TARGET_LINEAR_RESOLUTION * sqrt(2.0ff)] (the minimum and maximum radii in pixel coordinates).
- Let (M_x, M_y) be $\left(\frac{S_x^{max} - S_x^{min}}{R_{max} - R_{min}}, \frac{S_y^{max} - S_y^{min}}{R_{max} - R_{min}}\right)$
- Let A_{ratio} be the radial function aspect ratio defined as the length in pixels of the ellipse X axis over the Y axis.

Then determine SV_CoarsePixelSize as follows:

DeltaX = (C_x - P_x)

DeltaY = (C_y - P_y)

*If (A_{ratio} <= 1) DeltaY = A_{ratio} * DeltaY*

*Else DeltaX = (1/A_{ratio})*DeltaX*

D= Distance(DeltaX,DeltaY)

*$CoarsePixelSize_x$ = M_x * (D - R_{min}) + S_x^{min}*

$CoarsePixelSize_y$ = M_y(D- R_{min})+ S_y^{min}*

$CoarsePixelSize_x$ = clamp($CoarsePixelSize_x$, S_x^{min}, S_x^{max})

$CoarsePixelSize_y$ = clamp($CoarsePixelSize_y$, S_y^{min}, S_y^{max})

RequestedCoarsePixelSize = ($CoarsePixelSize_x$, Coars, e$PixelSize_y$)

Where *Distance()* is the Pythagorean distance function between two points defined by the following function:

$$Distance(DeltaX, DeltaY) = \sqrt{DeltaX^2 + DeltaY^2}$$

$$Distance(DeltaX, DeltaY) = \sqrt{DeltaX^2 + DeltaY^2}$$

However, because this function may be expensive to compute exactly, the distance function may be approximated by HW.

16 samples Per Coarse Pixel

Hardware supports a maximum of 16 sample per Coarse Pixel (CP).

Hardware will clamp any provided or computed CPsize to the nearest green CP size if not supported.

Table shows supported samples per CP as a function of CP size and MSAA rate.

- MSFT VRS spec (+Cap) requires the green filled entries.
- CPS_MODE_CONSTANT: Green + Orange entries
    - +orange enabled via setting pss_mode3::"Extended Coarse Pixel sizes"
- CPS_MODE_RADIAL : Green entries

Coarse pixel size	1x MSAA	2x MSAA	4x MSAA	8x MSAA	16x MSAA
1x2	2	4	8	16	32
2x1	2	4	8	16	32
2x2	4	4	16	32	64
2x4	8	16	32	64	64
4x2	8	16	32	64	128
4x4	16	32	64	128	256

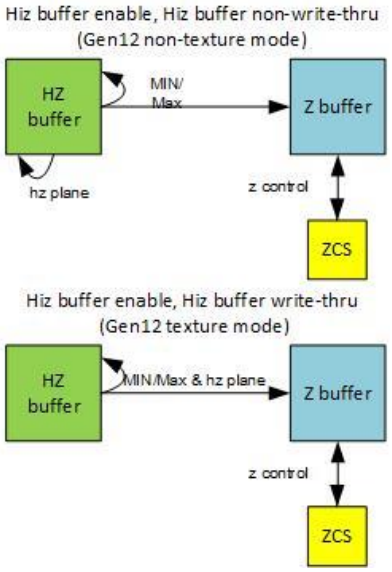
CPsize 1x4 and 4x1 are not supported

Early Depth/Stencil Processing

The Windower/IZ unit provides the Early Depth Test function, a major performance-optimization feature where an attempt is made to remove pixels that fail the Depth and Stencil Tests prior to pixel shading. This requires the WM unit to perform the interpolation of pixel ("source") depth values, read the current ("destination") depth values from the cached depth buffer, and perform the Depth and Stencil Tests As the WM unit has per-pixel source and destination Z values, these values are passed in the PS thread payload, if required.

Compressed Depth Buffer

The depth buffer can be compressed. 2 modes of operation are possible by using the 3DSTATE_DEPTH_BUFFER::Depth Buffer Compression Enable & the 3DSTATE_HIER_DEPTH_BUFFER:: Hierarchical Depth Buffer Write Thru Disable.



The operations of these 2 modes are given in the tables below.

	Non-texture performant Hiz Write thru == disabled Depth buffer compression == enabled			Texture performant Hiz Write thru == enabled Depth buffer compression == enabled		
	HiZ	ZCS	Z	HiZ	ZCS	Z
Min/Max cases	Update	Update on cache evict	Update Compress on cache evict	Update	Update on depth cache evict	Update Compress on cache evict
Plane cases	Update	ZCS buffer unchanged	Z buffer unchanged	Update	Update on depth cache evict	Update Compress on cache evict
Clears	Update	ZCS buffer unchanged	Z buffer unchanged	Update	Update with clear at either 16x8 or 8x4 granularity, based on fs_clr or otherwise	Update if entire cache line is not cleared Compress on cache evict

When the 3DSTATE_DEPTH_BUFFER::Depth Buffer Compression Enable is set, a compression control buffer must be allocated and the 3DSTATE_DEPTH_CNTL_BUFFER command is used to specify the buffer.

Depth Offset

Note: The depth offset function is contained in SF unit, thus the state to control it is also contained in SF unit.

There are occasions where the Z position of some objects need to be slightly offset to reduce artifacts due to coplanar or near-coplanar primitives. A typical example is drawing the edges of triangles as wireframes - the lines need to be drawn slightly closer to the viewer to ensure they will not be occluded by the underlying polygon. Another example is drawing objects on a wall - without a bias on the z positions, they might be fully or partially occluded by the wall.

The device supports *global* depth offset, applied only to triangles, that bases the offset on the object's z slope Note that there is no clamping applied at this stage after the Z position is offset - clamping to [0,1] can be performed later after the Z position is interpolated to the pixel. This is preferable to clamping prior to interpolation, as the clamping would change the Z slope of the entire object.

The Global Depth Offset function is controlled by the **Global Depth Offset Enable** state variable in WM_STATE Global Depth Offset is only applied to 3DOBJ_TRIANGLE objects.

When Global Depth Offset Enable is ENABLED, the pipeline will compute:

```
    MaxDepthSlope = max(abs(dZ/dX),abs(dz/dy)) // approximation of max depth slope for
polygon
```

When UNORM Depth Buffer is at Output Merger (or no Depth Buffer):

```
    Bias = GlobalDepthOffsetConstant * r + GlobalDepthOffsetScale * MaxDepthSlope
```

Where r is the minimum representable value > 0 in the depth buffer format, converted to float32 (note: If state bit **Legacy Global Depth Bias Enable** is set, the r term will be forced to 1.0)

When Floating Point Depth Buffer at Output Merger:

```
    Bias = GlobalDepthOffsetConstant * 2^(exponent(max z in primitive) - r) +
GlobalDepthOffsetScale * MaxDepthSlope
```

Where r is the # of mantissa bits in the floating point representation (excluding the hidden bit), e.g. 23 for float32 (note: If state bit Legacy Global Depth Bias Enable is set, no scaling is applied to the GobalDepthOffsetConstant).

Adding Bias to z:

```
    if (GlobalDepthOffsetClamp > 0)
        Bias = min(DepthBiasClamp, Bias)
    else if(GlobalDepthOffsetClamp < 0)
        Bias = max(DepthBiasClamp, Bias)
    // else if GlobalDepthOffsetClamp == 0, no clamping occurs
    z = z + Bias
```

Biasing is constant for a given primitive. The biasing formulas are performed with float32 arithmetic Global Depth Bias is not applied to any point or line primitives.

Early Depth Test/Stencil Test/Write

When **Early Depth Test Enable** is ENABLED, the WM unit will attempt to discard depth-occluded pixels during scan conversion (before processing them in the Pixel Shader). Pixels are only discarded when the WM unit can ensure that they would have no impact to the ColorBuffer or DepthBuffer. This function is therefore only a performance feature.

Note: The **Early Depth Test Enable** bit is no longer present. This function is always enabled.

If some pixels within a subspan are discarded, only the pixel mask is affected indicating that the discarded pixels are not active. If all pixels within a subspan are discarded, that subspan will not even be dispatched.

Software-Provided PS Kernel Info

For the WM unit to properly perform Early Depth Test and supply the proper information in the PS thread payload (and even determine if a PS thread needs to be dispatched), it requires information regarding the PS kernel operation This information is provided by a number of state bits in WM_STATE, as summarized in the following table.

State Bit	Description
Pixel Shader Kill Pixel	This must be set when there is a chance that valid pixels passed to a PS thread may be discarded. This includes the discard of pixels by the PS thread resulting from a "killpixel" or "alphatest" function or as dictated by the results of the sampling of a "chroma-keyed" texture The WM unit needs this information to prevent early depth/stencil writes for pixels which might be killed by the PS thread, etc. See WM_STATE/3DSTATE_WM for more information.
Pixel Shader Computed Depth	This must be set when the PS thread computes the "source" depth value (i.e., from the API POV, writes to the "oDepth" output) In this case the WM unit can't make any decisions based on the WM-interpolated depth value. See WM_STATE/3DSTATE_WM for more information.
Pixel Shader Uses Source Depth	Must be set if the PS thread requires the WM-interpolated source depth value. This forces the source depth to be passed in the thread payload where otherwise the WM unit would not have seen it as required. See WM_STATE/3DSTATE_WM for more information.

Hierarchical Depth Buffer

A hierarchical depth buffer is supported to reduce memory traffic due to depth buffer accesses.

3DSTATE_HIER_DEPTH_BUFFER_BODY lists what Tile Mode is supported.

The **Surface Type**, **Height**, **Width**, **Depth**, **Minimum Array Element**, **Render Target View Extent**, and **Depth Coordinate Offset X/Y** of the hierarchical depth buffer are inherited from the depth buffer. The height and width of the hierarchical depth buffer that must be allocated are computed by the following formulas, where HZ is the hierarchical depth buffer and Z is the depth buffer. The Z_Height, Z_Width, and Z_Depth values given in these formulas are those present in 3DSTATE_DEPTH_BUFFER incremented by one.

The Z_Height and Z_Width values must equal those present in 3DSTATE_DEPTH_BUFFER incremented by one.

Surface Type	HZ_Width (Bytes)	HZ_Height (Rows)	HZ_Qpitch (Rows)
SURFTYPE_1D			not applicable
SURFTYPE_2D	ceiling(Z_Width / 16) * 16	ceiling((HZ_QPitch/4)/16) *16 * Z_Depth	see below
SURFTYPE_3D			not applicable
SURFTYPE_CUBE	ceiling(Z_Width / 16) * 16	ceiling((HZ_QPitch/4)/16) *16 * 6 * Z_Depth	see below

To compute the minimum QPitch for the HZ surface, the height of each LOD in pixels is determined using the equations for hL in the GPU Overview volume, using a vertical alignment j=16. The following equation gives the minimum HZ_QPitch based on largest LOD m defined in the surface:

$$HZ\_QPitch = h_0 + max\left(h_1, \sum_{i=2}^{m} \mathrm{h_i}\right)$$

If m is less than 2, treat all h_L with L > m as zero and use the above equation.

The format of the data in the hierarchical depth buffer is not documented here, as this surface needs only to be allocated by software. Hardware will read and write this surface during operation and its contents are discarded once the last primitive is rendered that uses the hierarchical depth buffer.

The hierarchical depth buffer can be enabled whenever a depth buffer is defined, with its effect being invisible other than generally higher performance. The only exception is the hierarchical depth buffer must be disabled when using software tiled rendering.

If HiZ is enabled, you must initialize the clear value by either:

1. Perform a depth clear pass to initialize the clear value.
2. Send a 3dstate_clear_params packet with valid = 1.

Without one of these events, context switching will fail, as it will try to save off a clear value even though no valid clear value has been set. When context restore happens, HW will restore an uninitialized clear value.

Depth Buffer Clear

With the hierarchical depth buffer enabled, performance is generally improved by using the special clear mechanism described here to clear the hierarchical depth buffer and the depth buffer. This is enabled though the **Depth Buffer Clear** field in WM_STATE or 3DSTATE_WM or using the 3DSTATE_WM_HZ_OP. This bit can be used to clear the depth buffer in the following situations:

- Complete depth buffer clear.
- Partial depth buffer clear with the clear value the same as the one used on the previous clear.
- Partial depth buffer clear with the clear value different than the one used on the previous clear can use this mechanism if a depth buffer resolve is performed first.

The following is required when performing a depth buffer clear using any of the above clearing methods (WM_STATE, 3DSTATE_WM or 3DSTATE_WM_HZ_OP).

- The hierarchical depth buffer enable must be set in the 3DSTATE_DEPTH_BUFFER.
- The fields in 3DSTATE_CLEAR_PARAMS are set to indicate the source of the clear value and (if source is in this command) the clear value itself.
- The clear value must be between the min and max depth values (inclusive) defined in the CC_VIEWPORT. If the depth buffer.
    - The following alignment restrictions need to be met while doing the fast-clear:

Alignment Restriction
The minimum granularity of clear is one pixel, but all samples of the pixel must be cleared. Clearing partial samples of a pixel is not supported. If a newly allocated depth buffer is not padded to an integer multiple of 8x4 pixels, and if the first operation on the depth buffer does not clear the entire width and height of the surface, then first a HiZ ambiguate must be done on the portions of the depth buffer that are not cleared. If the depth buffer clear operation does clear the entire width and height of the surface, then the "full surface clear" bit in 3DSTATE_WM_OP must be set to 1.
The minimum granularity of clear is one pixel, but all samples of the pixel must be cleared. Clearing partial samples of a pixel is not supported. If the first operation of a newly allocated depth buffer does not clear the entire width and height of the surface, then first a HiZ ambiguate must be done on the portions of the depth buffer that are not cleared. If the depth buffer clear operation does clear the entire width and height of the surface, then the "full surface clear" bit in 3DSTATE_WM_OP must be set to 1.

The following is required when performing a depth buffer clear with using the WM_STATE or 3DSTATE_WM:

- If other rendering operations have preceded this clear, a PIPE_CONTROL with depth cache flush enabled, Depth Stall bit enabled must be issued before the rectangle primitive used for the depth buffer clear operation.
- **Depth Test Enable** must be disabled and **Depth Buffer Write Enable** must be enabled (if depth is being cleared).
- Stencil buffer clear can be performed at the same time by enabling Stencil Buffer Write Enable. Stencil Test Enable must be enabled and Stencil Pass Depth Pass Op set to REPLACE, and the clear value that is placed in the stencil buffer is the **Stencil Reference Value** from COLOR_CALC_STATE.
- Note also that stencil buffer clear can be performed without depth buffer clear. For stencil only clear, **Depth Test Enable** and **Depth Buffer Write Enable** must be disabled.

In some cases, **Depth Buffer Clear** cannot be enabled and the legacy method of clearing must be used:

- If the depth buffer format is D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT.
- If stencil test is enabled but the separate stencil buffer is disabled.

Depth buffer clear pass using any of the methods (WM_STATE, 3DSTATE_WM or 3DSTATE_WM_HZ_OP) must be followed by a PIPE_CONTROL command with DEPTH_STALL bit and Depth FLUSH bits "**set**" before starting to render.

Note: If using the optimized depth buffer clear, this pipecontrol should be done after the resetting of the clear/resolve bits in the 3DSTATE_WM_HZ_OP (step #8).

- Since the fast clear cycles to CCS are not cached in TileCache, any previous depth buffer writes to overlapping pixels must be flushed out of TileCache before a succeeding Depth Buffer Clear. This restriction only applies to Depth Buffer with write-thru enabled, since fast clears to CCS only occur for write-thru mode.

Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the depth buffer may contain incorrect results after rendering is complete If the depth buffer is retained and used for another purpose (i.e., as input to the sampling engine as a shadow map), it must first be "resolved" This is done by setting the **Depth Buffer Resolve Enable** field in WM_STATE or 3DSTATE_WM and rendering a full render target sized rectangle. Once this is complete, the depth buffer will contain the same contents as it would have had the rendering been performed with the hierarchical depth buffer disabled. In a typical usage model, depth buffer needs to be resolved after rendering on it and before using a depth buffer as a source for any consecutive operation. Depth buffer can be used as a source in three different cases: using it as a texture for the nest rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

The following is required when performing a depth buffer resolve:

- The surface must have been initialized with a Depth Buffer Clear after its allocation to initialize the Depth Clear Value.
- A rectangle primitive of the same size as the previous depth buffer clear operation must be delivered, and depth buffer state cannot have changed since the previous depth buffer clear operation.
- **Depth Test Enable** must be enabled with the **Depth Test Function** set to NEVER. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch, Alpha Test, Pixel Shader Kill Pixel** and **Pixel Shader Computed Depth** must all be disabled.

Programming Note	
Context:	HTML
HW uses the clear value from the 3DSTATE_CLEAR_PARAM. If you change the value in the 3DSTATE_CLEAR_PARAMS before resolve, it will flush the depth caches and have the new-clear value in its register. When doing the resolve pass, it is driver's responsibility to make sure that the clear-value for the depth buffer is the same one as the clear-pass.	

Hierarchical Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the hierarchical depth buffer may contain incorrect results if the depth buffer is written to outside of the 3D rendering operation If this occurs, the hierarchical depth buffer must be *resolved* to avoid incorrect device behavior. This is done by setting the Hierarchical Depth Buffer Resolve Enable field in WM_STATE or 3DSTATE_WM and rendering a full render target sized

rectangle. Once this is complete, the hierarchical depth buffer will contain contents such that rendering will give the same results as it would have had the rendering been performed with the hierarchical depth buffer disabled.

The following is required when performing a hierarchical depth buffer resolve:

- A rectangle primitive covering the full render target must be delivered.
- **Depth Test Enable** must be disabled. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch, Alpha Test, Pixel Shader Kill Pixel**, and **Pixel Shader Computed Depth** must all be disabled.

Optimized Depth Buffer Clear and/or Stencil Buffer Clear

With the hierarchical depth buffer enabled, performance is generally improved by using the special clear mechanism described here to clear the hierarchical depth buffer and the depth buffer This is enabled though the **Depth Buffer Clear** field in 3DSTATE_WM_HZ_OP This bit can be used to clear the depth buffer in the following situations:

- All 3D units before SF will be bypassed by WM_HZ_OP and states for those units need not be set/restored for these rectangles.
- Complete depth buffer clear
- Partial depth buffer clear with the clear value the same as the one used on the previous clear
- Partial depth buffer clear with the clear value different than the one used on the previous clear can use this mechanism if a depth buffer resolve is performed first.
- The minimum granularity of clear is one pixel, but all samples of the pixel must be cleared. Clearing partial samples of a pixel is not supported

Stencil Buffer Clears can be alone or at the same time as depth buffer clears by using the Stencil Buffer Clear bit in 3DSTATE_WM_HZ_OP.

Note for SURFACE_TYPE CUBE: To clear / resolve a CUBE_SURFACE using WM_HZ_OP, the surface_type must be changed to 2D and the depth is calculated for that.

As there are 6 faces of the cube, the depth is multiplied by 6 to get the number of slices in the cube. The min_array_index is one of the slices.

Hence, in order to clear / resolve, go through each slice & multiply depth by 6 and then using the min-array-index, point to the respective slice for clear/resolve.

The proper sequence of commands is as follows:

1. Setup 3DSTATE_DEPTH_BUFFER (as needed). Render Target Array index will be internally force to zero. SW must set 3DSTATE_DEPTH_BUFFER::**MinimumArrayElement** to render to the array to be cleared.
2. Setup 3DSTATE_HIER_DEPTH_BUFFER (as needed)
3. Setup 3DSTATE_STENCIL_BUFFER (as needed)

4. Setup 3DSTATE_CLEAR_PARMS (as needed).
5. Setup 3DSTATE_DRAWING_RECTANGLE (as needed and only if it is different from already existing drawing rectangle)
6. 3DSTATE_WM_HZ_OP w/ 1 of the clear/resolve bits set

   // This overrides existing state and forces them to what is needed for the clear

   // This also carries the vertex info for doing the clear
7. PIPE_CONTROL w/ all bits clear except for "Post-Sync Operation" must set to "Write Immediate Data" enabled.

   // This causes 3DSTATE_WM_HZ_OP state to be committed to SF and WM as a pipeline state. Once state is committed to SF, causes to spawn a rectangle to be drawn
8. 3DSTATE_WM_HZ_OP w/ none of the clear/resolve bits set

   // This clears the overrides
9. Restore 3DSTATE_DEPTH_BUFFER (as needed).
10. Restore 3DSTATE_HIER_DEPTH_BUFFER (as needed)
11. Restore 3DSTATE_STENCIL_BUFFER (as needed)

Arbitrary size rectangles are supported using the Top Left X, Top Left Y, Bottom Right X, Bottom Right Y fields in the 3DSTATE_WM_HZ_OP.

Optimized Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the depth buffer may contain incorrect results after rendering is complete If the depth buffer is retained and used for another purpose (i.e., locked by the app), it must first be "resolved" This is done by setting the **Depth Buffer Resolve Enable** field in 3DSTATE_WM_HZ_OP. The depth buffer resolve uses the same sequence as the optimized Depth buffer clear (see above) except the **Depth Buffer Resolve Enable** bit is set. Once this is complete, the depth buffer will contain the same contents as it would have had the rendering been performed with the hierarchical depth buffer disabled. In a typical usage model, depth buffer needs to be resolved after rendering on it and before using a depth buffer as a source for any consecutive operation. Depth buffer can be used as a source in three different cases: using it as a texture for the nest rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

Doing a resolve operation requires that a preceding Depth Buffer Clear operation is required to have initialized the Depth Clear Value.

Optimized Hierarchical Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the hierarchical depth buffer may contain incorrect results if the depth buffer is written to outside of the 3D rendering operation. If this occurs, the hierarchical depth buffer must be "resolved" to avoid incorrect device behavior. This is done by setting the **Hierarchical Depth Buffer Resolve Enable** field in 3DSTATE_WM_HZ_OP and specifying a full render target sized rectangle. The depth buffer resolve uses the same sequence as the optimized Depth buffer clear (see above) except the **Hierarchical Depth Buffer Resolve Enable** bit is set. Once this is complete, the

hierarchical depth buffer will contain contents such that rendering will give the same results as it would have had the rendering been performed with the hierarchical depth buffer disabled.

The following is required when performing a hierarchical depth buffer resolve:

- A rectangle primitive covering the full render target must be programmed on Xmin, Ymin, Xmax, and Ymax in the 3DSTATE_WM_HZ_OP command.
- The rectangle primitive size must be aligned to 8x4 pixels.

Separate Stencil Buffer

The following tables describe the separate stencil buffer for different generations.

The stencil buffer has a format of R8_UNIT,. The **Surface Type**, **Height**, **Width**, and **Depth**, **Minimum Array Element**, **Render Target View Extent**, **Depth Coordinate Offset X/Y**, **LOD**, and **Stencil Buffer Object Control State** are defined in the 3DSTATE_STENCIL_BUFFER

DepthStencil Buffer State

This section contains the state registers for the Depth/Stencil Buffers.

Register
3DSTATE_DEPTH_BUFFER
3DSTATE_STENCIL_BUFFER
3DSTATE_HIER_DEPTH_BUFFER
3DSTATE_CLEAR_PARAMS

Pixel Shader Thread Generation

After a group of object fragments have been rasterized, the Pixel Shader (PSD) function is invoked to further compute output information and cause results to be written to output surfaces (like color, depth, stencil, UAvs etc.). Fragments can be P or S.

Fragments can also be CP.

For each fragment, the Pixel Shader calculates the values of the various vertex attributes that are to be interpolated across the object using the interpolation coefficients. It then executes an API-supplied Pixel Shader Program. Instructions in this program permit the accessing of texture map data, where Texture Samplers are employed to sample and filter texture maps (see the Shared Functions chapter). Arithmetic operations can be performed on the texture data, input fragment information, and Pixel Shader Constants to compute the resultant fragment's output. The Pixel Shader program also allows the pixel to be discarded from further processing.

3DSTATE_PS

This command is used to set state used by the pixel shader dispatch stage.

Command
3DSTATE_PS

Programming Note	
Context:	Pixel Shader Thread Generation
Note: The PS Unit also receives 3DSTATE_PS_BLEND, 3DSTATE_SAMPLEMASK, 3DSTATE_MULTISAMPLE, and 3DSTATE_PS_EXTRA.	

Signal	PS_INT::oMask Present to RenderTarget
Description	This bit is inserted in the PS payload header and made available to the DataPort (either via the message header or via header bypass) to indicate that oMask data (one or two phases) is included in Render Target Write messages. If present, the oMask data is used to mask off samples.
Formula	= 3DSTATE_PS_EXTRA::oMask Present to RenderTarget

Signal	PS_INT::Dual Source Blend Enable
Description	This field is set if dual source blend is enabled. If this bit is disabled, the data port dual source message reverts to a single source message using source 0.
Formula	= 3DSTATE_PS_BLEND::ColorBufferBlendEnable && (PS_INT::UsesSrc1BlendFactor || (PS_INT::IndependentAlphaUsesSrc1BlendFactors && 3DSTATE_PS_BLEND::Independent Alpha Blend Enable))

Signal	PS_INT::UsesSrc1BlendFactor
Description	
Formula	= (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_SRC1_COLOR) ||

	(3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::DestinationBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA)

Signal	PS_INT::IndependentAlphaUsesSrc1BlendFactors
Description	
Formula	= (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::SourceAlphaBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_SRC1_COLOR) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_SRC1_ALPHA) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) || (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA)

Signal	PS_INT::PS UAV-only
Description	Pixel Shader UAV-only render target
Formula	= 3DSTATE_PS_EXTRA::Pixel Shader Has UAV && !3DSTATE_PS_EXTRA:: Pixel Shader Does not write to RT

Command
3DSTATE_PS_EXTRA This command is used to set state used by the pixel shader dispatch stage
3DSTATE_PS_BLEND This command is used to set state used by the pixel shader dispatch stage
3DSTATE_CONSTANT_PS
3DSTATE_BINDING_TABLE_POINTERS_PS
3DSTATE_PUSH_CONSTANT_ALLOC_PS
3DSTATE_SAMPLER_STATE_POINTERS_PS

Pixel Grouping (Dispatch Size) Control

The WM unit can pass a grouping of 2 subspans (8 pixels), 4 subspans (16 pixels), or 8 subspans (32 pixels) to a Pixel Shader thread. Software should take into account the following considerations when determining which groupings to support/enable during operation. This determination involves a tradeoff of these likely conflicting issues. Note that the size of the dispatch has significant impact on the kernel program. (It is certainly not transparent to the kernel.) Also note that there is no implied spatial relationship between the subspans passed to a PS thread, other than the fact that they come from the same object.

- **Thread Efficiency**: In general, there is some amount of overhead involved with PS thread dispatch, and if this can be amortized over a larger number of pixels, efficiency will likely increase. This is especially true for very short PS kernels, as may be used for desktop composition, etc.
- **GRF Consumption**: Processing more pixels per thread requires a larger thread payload and likely more temporary register usage, both of which translate into a requirement for a larger GRF register allocation for the threads. This increased GRF usage could lead to increased use of scratch space (for spill/fill, etc.) and possibly less efficient use of the EUs (as it would be less likely to find an EU with enough free physical GRF registers to service the thread).
- **Object Size**: If the number of very small objects (e.g., covering 2 subspans or fewer) is expected to comprise a significant portion of the workload, supporting the 8-pixel dispatch mode may be advantageous. Otherwise, there could be a large number of 16-pixel dispatches with only 1 or 2 valid subspans, resulting in low efficiency for those threads.
- **Intangibles**: Kernel footprint & Instruction Cache impact; Complexity; ....

The groupings of subspans that the WM unit is allowed to include in a PS thread payload is controlled by the **32,16,8 Pixel Dispatch Enable** state variables programmed in WM_STATE. Using these state variables, the WM unit attempts to dispatch the largest allowed grouping of subspans. The following table lists the possible combinations of these state variables.

Please note that, the valid column in the table indicates which products supports the combination dispatch. Combinations that are not listed in the table are not available on any product.

The letter codes A, B, D, and E used in the Variable Pixel Dispatch table below are valid for all projects with some specific mode restrictions for specific projects for B, D, and E as indicated in the next few tables.

D is like B with an added general restriction, that it cannot be used in non-1x PERSAMPLE mode.

E cannot be used in PERSAMPLE mode with number of multisamples >= 2.

See SIMD32 enable for additional programming restrictions.

F is valid combination.

Variable Pixel Dispatch

Contiguous 64 Pixel Dispatch Enable	Contiguous 32 Pixel Dispatch Enable	32 Pixel Dispatch Enable	16 Pixel Dispatch Enable	8 Pixel Dispatch Enable	Valid	IP for n-pixel Contiguous Dispatch		IP for n-pixel Dispatch (KSP offsets are in 128-bit instruction units)		
						n=64	n=32	n=32	n=16	n=8
0	0	0	0	1	A					KSP[0]
0	0	0	1	0	B				KSP[0]	
0	0	0	1	1	D				KSP[2]	KSP[0]
0	0	1	0	0	B			KSP[0]		
0	0	1	1	0	E			KSP[1]	KSP[2]	
0	0	1	1	1	D			KSP[1]	KSP[2]	KSP[0]
0	0	1	0	1	F			KSP[1]		KSP[0]
0	1	1	1	0	D		KSP[2]	KSP[1]	KSP[0]	
1	0	1	1	0	D	KSP[2]		KSP[1]	KSP[0]	

Each of the three KSP values is separately specified. In addition, each kernel has a separately specified GRF register count.

Depending on the subspan grouping selected, the WM unit will modify the starting PS Instruction Pointer (derived from the Kernel Start Pointer in WM_STATE) as a means to inform the PS kernel of the number of subspans included in the payload. The modified IP is a function of the enabled modes and the dispatch size, as shown in the table below.

HW will pick the right Kernel start pointer according to the dispatch size. (Note that the pointer from WM_STATE is 64-byte aligned which corresponds to four 128-bit instructions.)

If only one dispatch mode is enabled, the Jitter should not include any jump table entries at the beginning of the PS kernel. If multiple dispatch modes are enabled, a two-entry jump table should always be inserted, regardless of which modes are enabled (jump table entry for 8 pixel dispatch, followed by jump table entry for 32 pixel dispatch).

Note that for SIMD32 dispatch, pixel shader dispatch function increments GRF Start Register for URB Data state by 2 to account for the additional SIMD16 payload. The Pixel Shader kernel needs to comprehend this modification for SIMD32.

```
if ( 32PixelDispatchEnable && n > 7 )
    Dispatch 32 Pixels
else if ( 16PixelDispatchEnable && ( n > 2 || ! 8PixelDispatchEnable) )
    Dispatch 16 Pixels
else
    Dispatch 8 Pixels
end if
```

Multisampling Effects on Pixel Shader Dispatch

The pixel shader payloads are defined in terms of subspans and pixels. The slots in the pixel shader thread previously mapped 1:1 with pixels. With multisampling, a slot could contain a pixel or may just contain a single sample, depending on the mode. Payload definitions now refer to *slot* to make the definition independent of multisampling mode.

MSDISPMODE_PERPIXEL Thread Dispatch

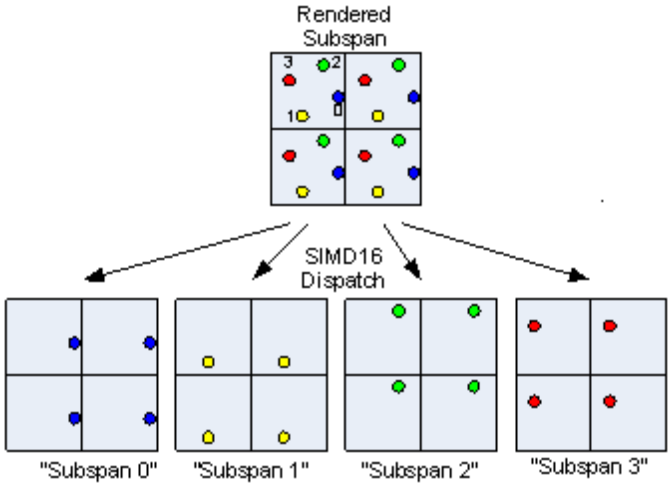
In PERPIXEL mode, the pixel shader kernel still works on 2/4/8 separate subspans, depending on dispatch mode. The fact that rasterization and the depth/stencil tests are being performed on a per-sample (not per-pixel) basis is transparent to the pixel shader kernel.

Programming Note	
Context:	MSDISPMODE_PERPIXEL Thread Dispatch
When NUM_MULTISAMPLES == 16 (i.e. 16x MSAA) and PS_DISPATCH_MODE is PER_PIXEL, SIMD32 pixel shader is not supported.	

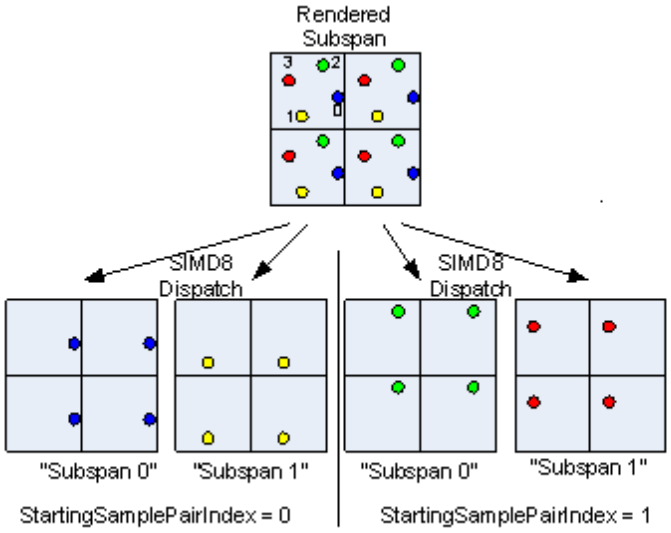
MSDISPMODE_PERSAMPLE Thread Dispatch

In PERSAMPLE mode, the pixel shader needs to operate on a sample vs. pixel basis (although this collapses in NUMSAMPLES_1 mode) Instead of processing strictly different subspans in parallel , the PS kernel processes different sample indices of one or more subspans in parallel For example, a SIMD16 dispatch in PERSAMPLE/NUMSAMPLES_4 mode would operate on a single subspan, with the usual "4 Subspan0 pixel slots" used for the "4 Sample0 locations of the (single) subspan" Subspan1 slots would be used for the Sample1 locations, and so on This layout allows the pixel shader to compute derivatives/LOD based on deltas between corresponding sample locations in the subspan in the same fashion as LEGACY pixel shader execution, and as required by DX10.1.

Depending on the dispatch mode (8/16/32 pixels) and multisampling mode (1X/4X), there are different mappings of subspans/samples onto dispatches and slots-within-dispatch In some cases, more than one subspan may be included in a dispatch, while in other cases multiple dispatches are be required to process all samples for a single subspan In the latter case, the **StartingSamplePairIndex** value is included in the payload header so the Render Target Write message will access the correct samples with each message.



PERSAMPLE SIMD16 4X Dispatch



PERSAMPLE Dispatch

The following table provides the complete dispatch/slot mappings for all the MS/Dispatch combinations.

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
SIMD32	*1X*	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]*

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		Slot[19:16] = Subspan[4].Pixel[3:0].Sample[0] *Slot[23:20] = Subspan[5].Pixel[3:0].Sample[0]* *Slot[27:24] = Subspan[6].Pixel[3:0].Sample[0]* *Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]*
	2X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1]* *Slot[19:16] = Subspan[2].Pixel[3:0].Sample[0]* *Slot[23:20] = Subspan[2].Pixel[3:0].Sample[1]* *Slot[27:24] = Subspan[3].Pixel[3:0].Sample[0]* *Slot[31:28] = Subspan[3].Pixel[3:0].Sample[1]*
	4X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]* *Slot[19:16] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[23:20] = Subspan[1].Pixel[3:0].Sample[1]* *Slot[27:24] = Subspan[1].Pixel[3:0].Sample[2]* *Slot[31:28] = Subspan[1].Pixel[3:0].Sample[3]*
	8X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]* *Slot[19:16] = Subspan[0].Pixel[3:0].Sample[4]* *Slot[23:20] = Subspan[0].Pixel[3:0].Sample[5]* *Slot[27:24] = Subspan[0].Pixel[3:0].Sample[6]* *Slot[31:28] = Subspan[0].Pixel[3:0].Sample[7]*

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
	16x	*Dispatch[i]: (i=0, 4)* *SSPI = i* *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]* *Slot[19:16] = Subspan[0].Pixel[3:0].Sample[SSPI*2+4]* *Slot[23:20] = Subspan[0].Pixel[3:0].Sample[SSPI*2+5]* *Slot[27:24] = Subspan[0].Pixel[3:0].Sample[SSPI*2+6]* *Slot[31:28] = Subspan[0].Pixel[3:0].Sample[SSPI*2+7]*
SIMD16	*1X*	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]*
	2X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[1].Pixel[3:0].Sample[0]* *Slot[15:12] = Subspan[1].Pixel[3:0].Sample[1]*
	4X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]*
	8X	*Dispatch[i]: (i=0, 2)* *SSPI = i* *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]*

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
	16x	*Dispatch[i]: (i=0, 2, 4, 6)* *SSPI = i* *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]* *Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2]* *Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]*
SIMD8	*1X*	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]*
	2X	*Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]*
	4X	*Dispatch[i]: (i=0..1)* *SSPI = i* *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]*
	8X	*Dispatch[i]: (i=0, 1, 2, 3)* *SSPI = i* *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]*
	16x	*Dispatch[i]: (i=0, 1, 2, 3, 4, 5, 6, 7)* *SSPI = i* *Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]* *Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]*

PS Thread Payload for Normal Dispatch

The following table lists all possible contents included in a PS thread payload, in the order they are provided. Certain portions of the payload are optional, in which case the corresponding phase is skipped.

PS Thread Payload for Normal Dispatch

All registers are numbered starting at 0, but many registers are skipped depending on configuration. This causes all registers below to be renumbered to fill in the skipped locations. The only case where actual registers may be skipped is immediately before the constant data and again before the setup data.

PS Thread Payload for Normal Dispatch

DWord	Bits	Description
R0.7	31	Reserved.
	30:24	Reserved
	23:0	**Primitive Thread ID:** This field contains the primitive thread count passed to the Windower from the Strips Fans Unit. Format: Reserved for HW Implementation Use.
R0.6	31:24	Reserved
	23:0	**Thread ID:** This field contains the fused thread count which is incremented by the Windower for every fused thread that is dispatched. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Buffer.** Specifies the index of the scratch buffer allocated to the stage. Format = SurfaceStateOffset[27:6]
	9:8	Reserved
	7:0	**FFTID:** This ID is assigned by the WM unit and is an identifier for the fused thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the Sampler State table. It is specified as an offset from the **Dynamic State Base Address**. Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	Reserved
R0.2	31:0	Reserved: Delivered as zeros (reserved for message header fields).
R0.1	31:6	**Color Calculator State Pointer:** Specifies the 64-byte aligned pointer to the Color Calculator

DWord	Bits	Description
		state (COLOR_CALC_STATE structure in memory). It is specified as an offset from the **Dynamic State Base Address**. This value is eventually passed to the ColorCalc function in the DataPort and is used to fetch the corresponding CC_STATE data. Format = DynamicStateOffset[31:5]
	5:0	Reserved
R0.0	31:14	Reserved
	13	**Source Depth to Render Target:** Indicates that source depth will be sent to the render target.
	12	**oMask to Render Target:** Indicates that oMask will be sent to the render target.
	11:5	Reserved
	4:0	**Primitive Topology Type:** This field identifies the Primitive Topology Type associated with the primitive spawning this object. The WM unit does not modify this value (e.g., objects within POINTLIST topologies see POINTLIST). Format: (See 3DPRIMITIVE command in *3D Pipeline*.)
R1.7	31:16	**Pixel/Sample Mask (SubSpan[3:0]):** Indicates which pixels within the four subspans are lit. **Note:** This is not a duplicate of the Dispatch Mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel/Sample Mask Copy (SubSpan[3:0]):** This is a duplicate copy of the pixel mask. This copy can be modified as the pixel shader thread executes in order to turn off pixels based on kill instructions.
R1.6	31	Reserved
	30:27	**Viewport Index 1:** Specifies the index of the viewport currently being used for upper SIMD8 of dual-SIMD8 thread. Format = U4 Range = [0,15]
	26:16	**Render Target Array Index 1:** Specifies the array index to be used for upper SIMD8 of dual-SIMD8 thread Following surface types: SURFTYPE_1D: specifies the array index Range = [0,2047] SURFTYPE_2D: specifies the array index Range = [0,2047] SURFTYPE_3D: specifies the "r" coordinate Range = [0,2047]

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>SURFTYPE_CUBE: specifies the face identifier Range = [0,5]
<table><tr><th>Face</th><th>Render Target Array Index</th></tr><tr><td>+x</td><td>0</td></tr><tr><td>-x</td><td>1</td></tr><tr><td>+y</td><td>2</td></tr><tr><td>-y</td><td>3</td></tr><tr><td>+z</td><td>4</td></tr><tr><td>-z</td><td>5</td></tr></table>Format = U11</td></tr>
<tr><td></td><td>15</td><td>Front/Back Facing Polygon 1: Determines whether the polygon 1 is front or back facing.
Used by the render cache to determine which stencil test state to use.
0: Front Facing
1: Back Facing</td></tr>
<tr><td></td><td>14:9</td><td>Reserved</td></tr>
<tr><td></td><td>8</td><td>Reserved for expansion of Starting Sample Pair Index 1</td></tr>
<tr><td></td><td>7:6</td><td>Starting Sample Pair Index 1: Indicates the index of the second sample pair of dual-SIMD8 dispatch.
Format = U2
Range = [0,3]</td></tr>
<tr><td></td><td>5:0</td><td>Reserved</td></tr>
<tr><td>R1.5</td><td>31:16</td><td>Y3: Y coordinate (screen space) for upper-left pixel of subspan 3 (slot 12).
Format = U16</td></tr>
<tr><td></td><td>15:0</td><td>X3: X coordinate (screen space) for upper-left pixel of subspan 3 (slot 12).
Format = U16</td></tr>
<tr><td>R1.4</td><td>31:16</td><td>Y2: Y coordinate (screen space) for upper-left pixel of subspan 2 (slot 8).
Format = U16</td></tr>
<tr><td></td><td>15:0</td><td>X2: X coordinate (screen space) for upper-left pixel of subspan 2 (slot 8).
Format = U16</td></tr>
<tr><td>R1.3</td><td>31:16</td><td>Y1: Y coordinate (screen space) for upper-left pixel of subspan 1 (slot 4).
Format = U16</td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td>15:0</td><td>X1: X coordinate (screen space) for upper-left pixel of subspan 1 (slot 4).
Format = U16</td></tr>
<tr><td>R1.2</td><td>31:16</td><td>Y0: Y coordinate (screen space) for upper-left pixel of subspan 0 (slot 0).
Format = U16</td></tr>
<tr><td></td><td>15:0</td><td>X0: X coordinate (screen space) for upper-left pixel of subspan 0 (slot 0).
Format = U16</td></tr>
<tr><td>R1.1</td><td>31</td><td>Reserved</td></tr>
<tr><td></td><td>30:27</td><td>Viewport Index 0: Specifies the index of the viewport currently being used.
Format = U4
Range = [0,15]</td></tr>
<tr><td></td><td>26:16</td><td>Render Target Array Index 0: Specifies the array index to be used for the following surface types:
SURFTYPE_1D: specifies the array index Range = [0,2047]
SURFTYPE_2D: specifies the array index Range = [0,2047]
SURFTYPE_3D: specifies the "r" coordinate Range = [0,2047]
SURFTYPE_CUBE: specifies the face identifier Range = [0,5]
<table>
<tr><th>Face</th><th>Render Target Array Index</th></tr>
<tr><td>+x</td><td>0</td></tr>
<tr><td>-x</td><td>1</td></tr>
<tr><td>+y</td><td>2</td></tr>
<tr><td>-y</td><td>3</td></tr>
<tr><td>+z</td><td>4</td></tr>
<tr><td>-z</td><td>5</td></tr>
</table>
Format = U11</td></tr>
<tr><td></td><td>15</td><td>Front/Back Facing Polygon 0: Determines whether the polygon 0 is front or back facing. Used by the render cache to determine which stencil test state to use.
0: Front Facing
1: Back Facing</td></tr>
<tr><td></td><td>14:9</td><td>Reserved</td></tr>
<tr><td></td><td>8</td><td>Reserved for expansion of Starting Sample Pair Index 0</td></tr>
</table>

DWord	Bits	Description
	7:6	**Starting Sample Pair Index 0:** Indicates the index of the first sample pair of the dispatch. Format = U2 Range = [0,3]
	5:0	Reserved
R1.0	31:20	Reserved
	19:16	**MSAA rate** (multisample count) Format: U4 [1..16] This field specifies MSAA sampling rate (required for PS+S monolithic shader).
	15:8	**ActualCoarsePixelShadingSize.Y** if coarse pixel dispatch Format: U8 This field specifies size (in pixels) of coarse pixel shading rate in Y dimension. Valid values are 1, 2, and 4. Note: coarse shading rate is constant for all coarse pixels in same thread dispatch.
	15:12	Slot 3 SampleID (if pixel or sample dispatch) Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
	11:8	Slot 2 SampleID (if pixel or sample dispatch) Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
	7:0	**ActualCoarsePixelShadingSize.X** if coarse pixel dispatch Format: U8 This field specifies size (in pixels) of coarse pixel shading rate in X dimension. Valid values are 1, 2, and 4. Note: coarse shading rate is constant for all coarse pixels in same thread dispatch.

DWord	Bits	Description
	7:4	Slot 1 SampleID (if pixel or sample dispatch) Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
	3:0	Slot 0 SampleID (if pixel or sample dispatch) Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7]
		R2: Delivered only if this is a *32-pixel dispatch*.
R2.7	31:16	**Pixel/Sample Mask (SubSpan[7:4]):** Indicates which pixels within the upper four subspans are lit. This field is valid only when the 32 pixel dispatch state is enabled. This field must not be modified by the pixel shader thread. Note: This is not a duplicate of the dispatch mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel/Sample Mask Copy (SubSpan[7:4]):** This is a duplicate copy of pixel mask for the upper 16 pixels. This copy will be modified as the pixel shader thread executes to turn off pixels based on kill instructions.
R2.6	31:0	Reserved
R2.5	31:16	**Y7:** Y coordinate (screen space) for upper-left pixel of subspan 7 (slot 28) Format = U16
	15:0	**X7:** X coordinate (screen space) for upper-left pixel of subspan 7 (slot 28) Format = U16
R2.4	31:16	Y6
	15:0	X6
R2.3	31:16	Y5

DWord	Bits	Description
	15:0	X5
R2.2	31:16	Y4
	15:0	X4
R2.1	31:0	Reserved
R2.0	31:16	Reserved
	15:12	Slot 7 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7] 16X MSAA range [0..15]
	11:8	Slot 6 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7] 16X MSAA range [0..15]
	7:4	Slot 5 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3] 8X MSAA range [0..7] 16X MSAA range [0..15]
	3:0	Slot 4 SampleID Format = U4 1X MSAA range: [0] 2X MSAA range [0,1] 4X MSAA range [0..3]

DWord	Bits	Description
		8X MSAA range [0..7] 16X MSAA range [0..15]
		R3-R26: Delivered only if the corresponding **Barycentric Interpolation Mode** bit is set. Register phases containing Slot 8-15 data are not delivered in *8-pixel dispatch* mode.
R3.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 7 This and the next register phase is only included if the corresponding enable bit in **Barycentric Interpolation Mode** is set. Format = IEEE_Float
R3.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 6
R3.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 5
R3.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 4
R3.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 3
R3.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 2
R3.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 1
R3.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 0
R4		Perspective Pixel Location Barycentric[2] for Slots 7:0
R5.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 15
R5.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 14
R5.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 13
R5.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 12
R5.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 11
R5.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 10
R5.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 9
R5.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 8
R6		Perspective Pixel Location Barycentric[2] for Slots 15:8
R7:10		Perspective Centroid Barycentric
R11:14		Perspective Sample Barycentric
R15:18		Linear Pixel Location Barycentric
R19:22		Linear Centroid Barycentric
R23:26		Linear Sample Barycentric
		R27: Delivered only if **Pixel Shader Uses Source Depth** is set.
R27.7	31:0	Interpolated Depth for Slot 7 Format = IEEE_Float This and the next register phase is only included if **Pixel Shader Uses Source Depth**

DWord	Bits	Description
		(WM_STATE) is set.
R27.6	31:0	Interpolated Depth for Slot 6
R27.5	31:0	Interpolated Depth for Slot 5
R27.4	31:0	Interpolated Depth for Slot 4
R27.3	31:0	Interpolated Depth for Slot 3
R27.2	31:0	Interpolated Depth for Slot 2
R27.1	31:0	Interpolated Depth for Slot 1
R27.0	31:0	Interpolated Depth for Slot 0
		R28: Delivered only if **Pixel Shader Uses Source Depth** is set and this is not an *8-pixel dispatch*.
R28.7	31:0	Interpolated Depth for Slot 15
R28.6	31:0	Interpolated Depth for Slot 14
R28.5	31:0	Interpolated Depth for Slot 13
R28.4	31:0	Interpolated Depth for Slot 12
R28.3	31:0	Interpolated Depth for Slot 11
R28.2	31:0	Interpolated Depth for Slot 10
R28.1	31:0	Interpolated Depth for Slot 9
R28.0	31:0	Interpolated Depth for Slot 8
		R29: Delivered only if **Pixel Shader Uses Source W** is set.
R29.7	31:0	Interpolated W for Slot 7 Format = IEEE_Float This and the next register phase are only included if **Pixel Shader Uses Source W** (WM_STATE) is set.
R29.6	31:0	Interpolated W for Slot 6
R29.5	31:0	Interpolated W for Slot 5
R29.4	31:0	Interpolated W for Slot 4
R29.3	31:0	Interpolated W for Slot 3
R29.2	31:0	Interpolated W for Slot 2
R29.1	31:0	Interpolated W for Slot 1
R29.0	31:0	Interpolated W for Slot 0
		R30: Delivered only if **Pixel Shader Uses Source W** is set and this is not an *8-pixel dispatch*.
R30.7	31:0	Interpolated W for Slot 15
R30.6	31:0	Interpolated W for Slot 14
R30.5	31:0	Interpolated W for Slot 13

DWord	Bits	Description
R30.4	31:0	Interpolated W for Slot 12
R30.3	31:0	Interpolated W for Slot 11
R30.2	31:0	Interpolated W for Slot 10
R30.1	31:0	Interpolated W for Slot 9
R30.0	31:0	Interpolated W for Slot 8
		R31: Delivered only if **Position XY Offset Select** is either POSOFFSET_CENTROID or POSOFFSET_SAMPLE.
R31.7	31:24	Position Offset Y for Slot 15 This field contains either the CENTROID or SAMPLE position offset for Y, depending on the state of **Position XY Offset Select**. Format = U4.4 For non-CP rate dispatch: Range = [0.0,1.0) For CP rate dispatch: Range = [0.0,4.0)
	23:16	Position Offset X for Slot 15 This field contains either the CENTROID or SAMPLE position offset for X, depending on the state of **Position XY Offset Select**. Format = U4.4 For non-CP rate dispatch: Range = [0.0,1.0) For CP rate dispatch: Range = [0.0,4.0)
	15:8	Position Offset Y for Slot 14
	7:0	Position Offset X for Slot 14
R31.6	31:24	Position Offset Y for Slot 13
	23:16	Position Offset X for Slot 13
	15:8	Position Offset Y for Slot 12
	7:0	Position Offset X for Slot 12
R31.5:4		Position Offset X/Y for Slot[11:8]
R31.3:2		Position Offset X/Y for Slot[7:4]
R31.1:0		Position Offset X/Y for Slot[3:0]
		R32: Delivered only if **Pixel Shader Uses Input Coverage Mask** is set.
R32.7	31:0	Input Coverage Mask for Slot 7 Format = U32 This and the next register phase is only included if **Pixel Shader Uses Input Coverage Mask** (3DSTATE_PS) is set.

DWord	Bits	Description
		Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R32.6	31:0	Input Coverage Mask for Slot 6 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R32.5	31:0	Input Coverage Mask for Slot 5 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R32.4	31:0	Input Coverage Mask for Slot 4 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R32.3	31:0	Input Coverage Mask for Slot 3 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R32.2	31:0	Input Coverage Mask for Slot 2 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R32.1	31:0	Input Coverage Mask for Slot 1 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R32.0	31:0	Input Coverage Mask for Slot 0 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
		R33: Delivered only if **Pixel Shader Uses Input Coverage Mask** is set and this is not an *8-pixel dispatch*.
R33.7	31:0	Input Coverage Mask for Slot 15 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R33.6	31:0	Input Coverage Mask for Slot 14 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.

DWord	Bits	Description
R33.5	31:0	Input Coverage Mask for Slot 13 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R33.4	31:0	Input Coverage Mask for Slot 12 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R33.3	31:0	Input Coverage Mask for Slot 11 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R33.2	31:0	Input Coverage Mask for Slot 10 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R33.1	31:0	Input Coverage Mask for Slot 9 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
R33.0	31:0	Input Coverage Mask for Slot 8 Fields encode sample Coverage Mask for kernels dispatched at pixel-rate or pixel Coverage Mask for kernels dispatched at coarse-rate.
		R34-R57: Delivered only if the corresponding **Barycentric Interpolation Mode** bit is set and this is a *32-pixel dispatch*.
R34.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 23 This and the next register phase is only included if the corresponding enable bit in **Barycentric Interpolation Mode** is set. Format = IEEE_Float
R34.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 22
R34.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 21
R34.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 20
R34.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 19
R34.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 18
R34.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 17
R34.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 16
R35		Perspective Pixel Location Barycentric[2] for Slots 23:16

DWord	Bits	Description
R36.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 31
R36.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 30
R36.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 29
R36.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 28
R36.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 27
R36.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 26
R36.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 25
R36.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 24
R37		Perspective Pixel Location Barycentric[2] for Slots 31:24
R38:41		Perspective Centroid Barycentric
R42:45		Perspective Sample Barycentric
R46:49		Linear Pixel Location Barycentric
R50:53		Linear Centroid Barycentric
R54:57		Linear Sample Barycentric
		R58-R59: Delivered only if **Pixel Shader Uses Source Depth** is set and this is a *32-pixel dispatch*.
R58.7	31:0	Interpolated Depth for Slot 23 Format = IEEE_Float This and the next register phase is only included if **Pixel Shader Uses Source Depth** (WM_STATE) bit is set.
R58.6	31:0	Interpolated Depth for Slot 22
R58.5	31:0	Interpolated Depth for Slot 21
R58.4	31:0	Interpolated Depth for Slot 20
R58.3	31:0	Interpolated Depth for Slot 19
R58.2	31:0	Interpolated Depth for Slot 18
R58.1	31:0	Interpolated Depth for Slot 17
R58.0	31:0	Interpolated Depth for Slot 16
R59.7	31:0	Interpolated Depth for Slot 31
R59.6	31:0	Interpolated Depth for Slot 30
R59.5	31:0	Interpolated Depth for Slot 29
R59.4	31:0	Interpolated Depth for Slot 28
R59.3	31:0	Interpolated Depth for Slot 27
R59.2	31:0	Interpolated Depth for Slot 26
R59.1	31:0	Interpolated Depth for Slot 25
R59.0	31:0	Interpolated Depth for Slot 24

DWord	Bits	Description
		R60-R61: Delivered only if **Pixel Shader Uses Source W** is set and this is a *32-pixel dispatch*.
R60.7	31:0	Interpolated W for Slot 23 Format = IEEE_Float This and the next register phase are only included if **Pixel Shader Uses Source W** (WM_STATE) bit is set.
R60.6	31:0	Interpolated W for Slot 22
R60.5	31:0	Interpolated W for Slot 21
R60.4	31:0	Interpolated W for Slot 20
R60.3	31:0	Interpolated W for Slot 19
R60.2	31:0	Interpolated W for Slot 18
R60.1	31:0	Interpolated W for Slot 17
R60.0	31:0	Interpolated W for Slot 16
R61.7	31:0	Interpolated W for Slot 31
R61.6	31:0	Interpolated W for Slot 30
R61.5	31:0	Interpolated W for Slot 29
R61.4	31:0	Interpolated W for Slot 28
R61.3	31:0	Interpolated W for Slot 27
R61.2	31:0	Interpolated W for Slot 26
R61.1	31:0	Interpolated W for Slot 25
R61.0	31:0	Interpolated W for Slot 24
		R62: Delivered only if **Position XY Offset Select** is either POSOFFSET_CENTROID or POSOFFSET_SAMPLE and this is a *32-pixel dispatch*.
R62.7	31:24	Position Offset Y for Slot 31 This field contains either the CENTROID or SAMPLE position offset for Y, depending on the state of **Position XY Offset Select**. Format = U4.4 Range = [0.0,1.0)
	23:16	Position Offset X for Slot 31 This field contains either the CENTROID or SAMPLE position offset for X, depending on the state of **Position XY Offset Select**. Format = U4.4 Range = [0.0,1.0)
	15:8	Position Offset Y for Slot 30
	7:0	Position Offset X for Slot 30

DWord	Bits	Description
R62.6	31:24	Position Offset Y for Slot 29
	23:16	Position Offset X for Slot 29
	15:8	Position Offset Y for Slot 28
	7:0	Position Offset X for Slot 28
R62.5:4		Position Offset X/Y for Slot[27:24]
R62.3:2		Position Offset X/Y for Slot[23:20]
R62.1:0		Position Offset X/Y for Slot[19:16]
		R63-R64: Delivered only if **Pixel Shader Uses Input Coverage Mask** is set and this is a *32-pixel dispatch*.
R63.7	31:0	Input Coverage Mask for Slot 23 Format = U32 This and the next register phase are only included if **Pixel Shader Uses Input Coverage Mask** (3DSTATE_PS) is set.
R63.6	31:0	Input Coverage Mask for Slot 22
R63.5	31:0	Input Coverage Mask for Slot 21
R63.4	31:0	Input Coverage Mask for Slot 20
R63.3	31:0	Input Coverage Mask for Slot 19
R63.2	31:0	Input Coverage Mask for Slot 18
R63.1	31:0	Input Coverage Mask for Slot 17
R63.0	31:0	Input Coverage Mask for Slot 16
R64.7	31:0	Input Coverage Mask for Slot 31
R64.6	31:0	Input Coverage Mask for Slot 30
R64.5	31:0	Input Coverage Mask for Slot 29
R64.4	31:0	Input Coverage Mask for Slot 28
R64.3	31:0	Input Coverage Mask for Slot 27
R64.2	31:0	Input Coverage Mask for Slot 26
R64.1	31:0	Input Coverage Mask for Slot 25
R64.0	31:0	Input Coverage Mask for Slot 24
		R65 delivered ONLY if **Pixel Shader Requires RequiredCoarsePixelShadingSize** is set. Value is undefined for SIMD32 thread payload.
R65.7	31:0	**RequestedCoarsePixelShadingRate.Y** for subspan 3 (slot 12) This is post-clamp value, expected range [1.0f,4.0f] or inner range if min/max configured. Format: IEEE_Float
R65.6	31:0	**RequestedCoarsePixelShadingRate.Y** for subspan 2 (slot 8)

DWord	Bits	Description
		Format: IEEE_Float
R65.5	31:0	**RequestedCoarsePixelShadingRate.Y** for subspan 1 (slot 4) Format: IEEE_Float
R65.4	31:0	**RequestedCoarsePixelShadingRate.Y** for subspan 0 (slot 0) Format: IEEE_Float
R65.3	31:0	**RequestedCoarsePixelShadingRate.X** for subspan 3 (slot 12) Format: IEEE_Float
R65.2	31:0	**RequestedCoarsePixelShadingRate.X** for subspan 2 (slot 8) Format: IEEE_Float
R65.1	31:0	**RequestedCoarsePixelShadingRate.X** for subspan 1 (slot 4) Format: IEEE_Float
R65.0	31:0	**RequestedCoarsePixelShadingRate.X** for subspan 0 (slot 0) Format: IEEE_Float
	R66: delivered only if **Pixel Shader Requires Source Depth and/or W Attribute Vertex Deltas** is set.	
R66.7	31:0	**rhw_c0** – Co for 1/w plane Format = IEEE_Float
R66.6	31:0	**YStart** coordinate (screen space) for upper-left vertex of a triangle being rasterized. Format = float32
R66.5	31:0	**rhw_cx** – Cx for 1/w plane Format = IEEE_Float
R66.4	31:0	**rhw_cy** - Cy for 1/w plane Format = IEEE_Float
R66.3	31:0	**z_c0** – Co for z plane Format = IEEE_Float

DWord	Bits	Description
R66.2	31:0	**XStart** coordinate (screen space) for upper-left vertex of a triangle being rasterized. Format = float32
R66.1	31:0	**z_cx** – Cx for z plane Format = IEEE_Float
R66.0	31:0	**z_cy** - Cy for z plane Format = IEEE_Float
	R67: delivered only if **Pixel Shader Requires Source Depth and/or W Attribute Vertex Deltas** and **Dual 8-pixel dispatch** are set.	
R67		Defines RHW and Z coefficients for poly 1.
	R68: delivered only if **Pixel Shader Requires Perspective Bary Planes** is set.	
R68.7	31:0	**bary2_c0** – Co for bary2/w plane Format = IEEE_Float
R68.6	31:0	**YStart** coordinate (screen space) for upper-left vertex of a triangle being rasterized. Format = float32
R68.5	31:0	**bary2_cx** – Cx for bary2/w plane Format = IEEE_Float
R68.4	31:0	**bary2_cy** - Cy for bary2/w plane Format = IEEE_Float
R68.3	31:0	**bary1_c0** – Co for bary1/w plane Format = IEEE_Float
R68.2	31:0	**XStart** coordinate (screen space) for upper-left vertex of a triangle being rasterized. Format = float32
R68.1	31:0	**bary1_cx** – Cx for bary1/w plane Format = IEEE_Float
R68.0	31:0	**bary1_cy** - Cy for bary1/w plane Format = IEEE_Float

DWord	Bits	Description
	R69: delivered only if **Pixel Shader Requires Perspective Bary Planes** and **Dual 8-pixel dispatch** are set.	
R69		Defines perspective bary1/bary2 coefficients and vertex StartX/StartY for poly 1.
	R70: delivered only if **Pixel Shader Requires Non-Perspective Bary Planes** is set.	
R70.7	31:0	**npc_bary2_c0** – Co for npc_bary2 plane Format = IEEE_Float
R70.6	31:0	**YStart** coordinate (screen space) for upper-left vertex of a triangle being rasterized. Format = float32
R70.5	31:0	**npc_bary2_cx** – Cx for npc_bary2 plane Format = IEEE_Float
R70.4	31:0	**npc_bary2_cy** - Cy for npc_bary2 plane Format = IEEE_Float
R70.3	31:0	**npc_bary1_c0** – Co for npc_bary1 plane Format = IEEE_Float
R70.2	31:0	**XStart** coordinate (screen space) for upper-left vertex of a triangle being rasterized. Format = float32
R70.1	31:0	**npc_bary1_cx** – Cx for npc_bary1 plane Format = IEEE_Float
R70.0	31:0	**npc_bary1_cy** - Cy for npc_bary1 plane Format = IEEE_Float
	R71: delivered only if **Pixel Shader Requires Non-Perspective Bary Planes** and **Dual 8-pixel dispatch** are set.	
R71		Defines non-perspective bary1/bary2 coefficients and vertex StartX/StartY for poly 1.
	R72: delivered only if **Pixel Shader Requires sample offsets** is set.	
R72.7	31:28	Reserved – MBZ
	29:24	Sub-sample Y offset for sample 15 Format: U2.4

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>Sub-pixel Y offset of Sample 15 relative to the UL pixel origin if not coarse pixel dispatch

Sub-coarsepixel Y offset of Sample 15 relative to the UL coarse pixel origin if coarse pixel dispatch

Number of bits and range is based on ActualCoarsePixelShadingSize.Y if coarse pixel dispatch

<table>
<tr><td>Dispatch Rate</td><td>CPsize.Y</td><td>Format</td><td>Range</td></tr>
<tr><td>sample or pixel</td><td>---</td><td>U0.4</td><td>[0,0.9375] (0/16 - 15/16)</td></tr>
<tr><td>coarse</td><td>1</td><td>U0.4</td><td>[0,0.9375] (0/16 - 15/16)</td></tr>
<tr><td>coarse</td><td>2</td><td>U1.4</td><td>[0,1.9375] (0/16 - 31/16)</td></tr>
<tr><td>coarse</td><td>4</td><td>U2.4</td><td>[0,3.9375] (0/16 - 63/16)</td></tr>
</table>

The number of valid sub-sample offsets is the number of MSAA samples if not coarse pixel dispatch

The number of valid sub-sample offsets is CPsize.X * CPsize.Y * num_MSAA_samples if coarse pixel dispatch</td></tr>
<tr><td></td><td>23:22</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>21:16</td><td>Sub-sample Y offset for sample 14</td></tr>
<tr><td></td><td>15:14</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>13:8</td><td>Sub-sample Y offset for sample 13</td></tr>
<tr><td></td><td>7:6</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>5:0</td><td>Sub-sample Y offset for sample 12</td></tr>
<tr><td>R72.6</td><td>31:30</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>29:24</td><td>Sub-sample Y offset for sample 11</td></tr>
<tr><td></td><td>23:22</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>21:16</td><td>Sub-sample Y offset for sample 10</td></tr>
<tr><td></td><td>15:14</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>13:8</td><td>Sub-sample Y offset for sample 9</td></tr>
<tr><td></td><td>7:6</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>5:0</td><td>Sub-sample Y offset for sample 8</td></tr>
<tr><td>R72.5</td><td>31:30</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>29:24</td><td>Sub-sample Y offset for sample 7</td></tr>
<tr><td></td><td>23:22</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>21:16</td><td>Sub-sample Y offset for sample 6</td></tr>
<tr><td></td><td>15:14</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>13:8</td><td>Sub-sample Y offset for sample 5</td></tr>
<tr><td></td><td>7:6</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>5:0</td><td>Sub-sample Y offset for sample 4</td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td>R72.4</td><td>31:30</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>29:24</td><td>Sub-sample Y offset for sample 3</td></tr>
<tr><td></td><td>23:22</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>21:16</td><td>Sub-sample Y offset for sample 2</td></tr>
<tr><td></td><td>15:14</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>13:8</td><td>Sub-sample Y offset for sample 1</td></tr>
<tr><td></td><td>7:6</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>5:0</td><td>Sub-sample Y offset for sample 0</td></tr>
<tr><td>R72.3</td><td>31:30</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>29:24</td><td>Sub-sample X offset for sample 15

Format: U2.4

Sub-pixel X offset of Sample 15 relative to the UL pixel origin if not coarse pixel dispatch

Sub-coarsepixel X offset of Sample 15 relative to the UL coarse pixel origin if coarse pixel dispatch

Number of bits and range is based on ActualCoarsePixelShadingSize.X if coarse pixel dispatch

<table><tr><td>Dispatch Rate</td><td>CPsize.X</td><td>Format</td><td>Range</td></tr><tr><td>sample or pixel</td><td>---</td><td>U0.4</td><td>[0,0.9375] (0/16 - 15/16)</td></tr><tr><td>coarse</td><td>1</td><td>U0.4</td><td>[0,0.9375] (0/16 - 15/16)</td></tr><tr><td>coarse</td><td>2</td><td>U1.4</td><td>[0,1.9375] (0/16 - 31/16)</td></tr><tr><td>coarse</td><td>4</td><td>U2.4</td><td>[0,3.9375] (0/16 - 63/16)</td></tr></table>
The number of valid sub-sample offsets is the number of MSAA samples if not coarse pixel dispatch

The number of valid sub-sample offsets is CPsize.X * CPsize.Y * num_MSAA_samples if coarse pixel dispatch</td></tr>
<tr><td></td><td>23:22</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>21:16</td><td>Sub-sample X offset for sample 14</td></tr>
<tr><td></td><td>15:14</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>13:8</td><td>Sub-sample X offset for sample 13</td></tr>
<tr><td></td><td>7:6</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>5:0</td><td>Sub-sample X offset for sample 12</td></tr>
<tr><td>R72.2</td><td>31:30</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>29:24</td><td>Sub-sample X offset for sample 11</td></tr>
<tr><td></td><td>23:22</td><td>Reserved – MBZ</td></tr>
<tr><td></td><td>21:16</td><td>Sub-sample X offset for sample 10</td></tr>
<tr><td></td><td>15:14</td><td>Reserved – MBZ</td></tr>
</table>

DWord	Bits	Description
	13:8	Sub-sample X offset for sample 9
	7:6	Reserved – MBZ
	5:0	Sub-sample X offset for sample 8
R72.1	31:30	Reserved – MBZ
	29:24	Sub-sample X offset for sample 7
	23:22	Reserved – MBZ
	21:16	Sub-sample X offset for sample 6
	15:14	Reserved – MBZ
	13:8	Sub-sample X offset for sample 5
	7:6	Reserved – MBZ
	5:0	Sub-sample X offset for sample 4
R72.0	31:30	Reserved – MBZ
	29:24	Sub-sample X offset for sample 3
	23:22	Reserved – MBZ
	21:16	Sub-sample X offset for sample 2
	15:14	Reserved – MBZ
	13:8	Sub-sample X offset for sample 1
	7:6	Reserved – MBZ
	5:0	Sub-sample X offset for sample 0
R73.7	31:0	Primitive **Full Pixel Coverage** bitmask Bit *i*, when enabled, indicates an input pixel *i* is fully covered by the Xmin/Xmax/Ymin/Ymax bounding box. For example, with SIMD16 pixel shader dispatch mode, a value of 0x00FF indicates all 16 input pixels have their area fully covered by the input primitive bounding box. A pixel shader kernel can use this bitmask as fast check of full coverage condition. Format: U32
R73.4:6	31:0	Reserved - MBZ
R73.3	31:24	Reserved - MBZ
	23:0	Primitive **Ymax -** maximum value of sub-pixel aligned position.Y attribute in screen-space, computed from vertices forming an input primitive. Format: U16.8
R73.2	31:24	Reserved - MBZ
	23:0	Primitive **Ymin -** minumum value of sub-pixel aligned position.Y attribute in screen-space, computed from vertices forming an input primitive Format: U16.8
R73.1	31:24	Reserved - MBZ
	23:0	Primitive **Xmax -** maximum value of sub-pixel aligned position.X attribute in screen-space,

DWord	Bits	Description
		computed from vertices forming an input primitive Format: U16.8
R73.0	31:24	Reserved - MBZ
	23:0	Primitive **Xmin -** minimum value of sub-pixel aligned position.X attribute in screen-space, computed from vertices forming an input primitive Format: U16.8
R74		Defines sub-pixel aligned quad and full pixel coverage mask for poly 1.
		Optional Padding before the Start of Constant/Setup Data The locations between the end of the Optional Payload Header and the location programmed via Dispatch GRF Start Register for Constant/Setup Data are considered "padding" and Reserved (see below).
Optional, multiple of 8 DWs	31:0	Reserved
		The **Dispatch GRF Start Register for Constant/Setup Data** state variable in 3DSTATE_WM is used to define the starting location of the constant and setup data within the PS thread payload. This control is provided to allow this data to be located at a fixed location within thread payloads, regardless of the amount of data in the Optional Payload Header. This permits the kernel to use direct GRF addressing to access the constant/setup data, regardless of the optional parameters being passed (as these are determined on-the-fly by the WM unit).
[Varies] optional	255:0	**Constant Data (optional):** For more details about the size and source of constant data, please refer to **General Programming of Thread-Generating Stages** in the Push Constants chapter.
[Varies] optional	255:0	**Per-primitive Constant Data (optional):** Per-Primitive Attributes when Mesh Shader is enabled and Mesh Shader outputs per-primitive attributes. The number of GRF is indicated by **SBE_MESH::Per-Primitive URB Entry Output Read Length**. PolyConst.n = Rpc + n, Where Rpc is the first GRF with primitive-constant data. For **Dual-SIMD8** dispatch, per-primitive attribute data interleaved on per 32B (GRF register) granularity, i.e., 32B of Prim0 and followed by 32B of Prim1. Poly0Const.n = Rpc + 2*n. Poly1Const.n = Rpc + 2*n + 1
		Setup Data (Attribute Vertex Deltas) for dispatches other than dual 8-pixel dispatch. Output data from the SF stage is delivered in the PS thread payload. The amount of data is determined by the **Number of Output Attributes** field. Each register contains two channels

DWord	Bits	Description
		of one attribute. Thus, the total number of registers sent is equal to 2 * Number of Output Attributes.
Rp.7	31:0	**a0[0].y** – a0 vertex data for Attribute0.y
Rp.6	31:0	Reserved
Rp.5	31:0	**a2[0].y** – a2-a0 vertex delta for Attribute0.y
Rp.4	31:0	**a1[0].y** – a1-a0 vertex delta for Attribute0.y
Rp.3	31:0	**a0[0].x** – a0 vertex data for Attribute0.x
Rp.2	31:0	Reserved
Rp.1	31:0	**a2[0].x** – a2-a0 vertex delta for Attribute0.x
Rp.0	31:0	**a1[0].x** – a1-a0 vertex delta for Attribute0.x
R(p+1).7	31:0	**a0[0].w** – a0 vertex data for Attribute0.w
R(p+1).6	31:0	Reserved
R(p+1).5	31:0	**a2[0].w** – a2-a0 vertex delta for Attribute0.w
R(p+1).4	31:0	**a1[0].w** – a1-a0 vertex delta for Attribute0.w
R(p+1).3	31:0	**a0[0].z** – a0 vertex data for Attribute0.z
R(p+1).2	31:0	Reserved
R(p+1).1	31:0	**a2[0].z** – a2-a0 vertex delta for Attribute0.z
R(p+1).0	31:0	**a1[0].z** – a1-a0 vertex delta for Attribute0.z
R(p+2*n)		**xy Vertex Deltas for Attributes n** See definition of Rp for format.
R(p+2*n+1)		**zw Vertex Deltas for Attribute n** See definition of R(p+1) for format.
		Setup Data (Attribute Vertex Deltas) for dual 8-pixel dispatch. Output data from the SF stage is delivered in the PS thread payload. The amount of data is determined by the **Number of Output Attributes** field. Each register contains two channels of one attribute and same layout is used as for dispatches other than dual 8-pixel dispatch. However, two registers used - one for object 0 and another for object 1. Thus, the total number of registers sent is equal to 4 * Number of Output Attributes.
R(p+4*n)		**xy Vertex Deltas for Attribute n from object 0** See definition of Rp for format.
R(p+4*n+1)		**xy Vertex Deltas for Attribute n from object 1** See definition of Rp for format.
R(p+4*n+2)		**zw Vertex Deltas for Attribute n from object 0**

DWord	Bits	Description
		See definition of R(p+1) for format.
R(p+4*n+3)		**zw Vertex Deltas for Attribute n from object 1** See definition of R(p+1) for format.

Pixel Backend

This section contains the following subsections:

- MCS Buffer for Render Target(s)
- Render Target Fast Clear
- Render TargetResolve

Color Calculator (Output Merger)

Overview

Note: The Color Calculator logic resides in the Render Cache backing Data Port (DAP) shared function. It is described in this chapter as the Color Calc functions are naturally an extension of the 3D pipeline past the WM stage. See the DataPort chapter for details on the messages used by the Pixel Shader to invoke Color Calculator functionality.

The *Color Calculator* (referred to as "Output Merger in the DX Spec) function within the Data Port shared function completes the processing of rasterized pixels after the pixel color and depth have been computed by the Pixel Shader. This processing is initiated when the pixel shader thread sends a Render Target Write message (see *Shared Functions*) to the Render Cache. (Note that a single pixel shader thread may send multiple Render Target Write messages, with the result that multiple render targets get updated.) The pixel variables pass through a pipeline of fixed (yet programmable) functions, and the results are conditionally written into the appropriate buffers.

The word "pixel" used in this section is effectively replaced with the word "sample" if multisample rasterization is enabled.

Pipeline Stage	Description
Alpha Coverage	It generates coverage masks using AlphaToCoverage AND/OR AlphaToOne functions based on src0.alpha.
Alpha Test	Compare pixel alpha with reference alpha and conditionally discard pixel.
Stencil Test	Compare pixel stencil value with reference and forward result to Buffer Update stage.
Depth Test	Compare pix.Z with corresponding Z value in the Depth Buffer and forward result to Buffer Update stage.
Color Blending	Combine pixel color with corresponding color in color buffer according to programmable function.
Gamma Correction	Adjust pixel's color according to gamma function for SRGB destination surfaces.

Pipeline Stage	Description
Color Quantization	Convert "full precision" pixel color values to fixed precision of the color buffer format.
Logic Ops	Combine pixel color logically with existing color buffer color (mutually exclusive with Color Blending).
Buffer Update	Write final pixel values to color and depth buffers or discard pixel without update.

The following logic describes the high-level operation of the Pixel Processing pipeline:

```
PixelProcessing() {
    AlphaCoverage()
    AlphaTest()
    DepthBufferCoordinateOffsetDisable
    StencilTest()
    DepthTest()
    ColorBufferBlending()
    GammaCorrection()
    ColorQuantization()
    LogicalOps()
    BufferUpdate()
}
```

Alpha Coverage

Alpha coverage logic can be controlled using three state variables:

- **AlphaToCoverage Enable**, when enabled Color Calculator modifies the sample mask. This function (along with AlphaToOne) come at the top of the pixel pipeline. The sample's Source0.Alpha value (possibly being replicated from the pixel's Source0.Alpha) is used to compute a (optionally dithered) 1/2/4-bit mask (depending on NumSamples).
- The **AlphaToCoverage Dither Enable** SV is used to control the dithering of the AlphaToCoverage mask. The bit corresponding to the sample# is then ANDed with the sample's incoming mask bits - allowing the sample to be masked off depending on alpha.
- **AlphaToOne Enable,** when enabled, Color Calculator must replace Source0.Alpha (if present) with 1.0f.
- If AlphaToCoverage is disabled, AlphaToCoverage Dither does not have any impact.
- If Pixel Shader outputs oMask, AlphaToCoverage is disabled in hardware, regardless of the state setting for this feature.

Notes:

- Src0.alpha needs to be first multiplied with AA alpha before applying AlphaToCoverage and AlphaToOne functions.
- An alpha value of NaN results in a no coverage (zero) mask.
- Alpha values from the pixel shader are treated as FLOAT32 format for computing the AlphaToCoverage Mask.

Alpha Test

The Alpha Test function can be used to discard pixels based on a comparison between the incoming pixel's alpha value and the **Alpha Test Reference** state variable in COLOR_CALC_STATE. This operation can be used to remove transparent or nearly transparent pixels, though other uses for the alpha channel and alpha test are certainly possible.

This function is enabled by the **Alpha Test Enable** state variable in COLOR_CALC_STATE. If ENABLED, this function compares the incoming pixel's alpha value (*pixColor.Alpha*) and the reference alpha value specified by via the **Alpha Test Reference** state variable in COLOR_CALC_STATE. The comparison performed is specified by the **Alpha Test Function** state variable in COLOR_CALC_STATE.

The **Alpha Test Format** state variable is used to specify whether Alpha Test is performed using fixed-point (UNORM8) or FLOAT32 values. Accordingly, it determines whether the **Alpha Reference Value** is passed in a UNORM8 or FLOAT32 format. If UNORM8 is selected, the pixel's alpha value will be converted from floating-point to UNORM8 before the comparison.

Pixels that pass the Alpha Test proceed for further processing. Those that fail are discarded at this point in the pipeline.

If **Alpha Test Enable** is DISABLED, this pipeline stage has no effect.

The Alpha Test function is supported in conjunction with Multiple Render Targets (MRTs). If delivered in the incoming render target write message, source 0 alpha is used to perform the alpha test. If source 0 alpha is not delivered, the normal alpha value is used to perform the alpha test.

Depth Coordinate Offset

The Depth Coordinate Offset function applies a programmable constant offset to the RenderTarget X,Y screen space coordinates in order to generate DepthBuffer coordinates.

The function has been specifically added to allow the OpenGL driver to deal with a RenderTarget and DepthBuffer of differing sizes.

This condition is not an issue for the D3D driver, as D3D defines an upper-left screen coordinate origin which matches the HW rasterizer; as long as the application limits rendering to the smaller of the RT/DepthBuffer extents, no special logic is required.

OpenGL defines a lower-left screen coordinate origin. This requires the driver to incorporate a "Y coordinate flipping" transformation into the viewport mapping function. The Y extent of the RT is used in this flipping transformation. If the DepthBuffer extent is different, the wrong pixel Y locations within the DepthBuffer will be accessed.

The least expensive solution is to provide a translation offset to be applied to the post-viewport-mapped DepthBuffer Y pixel coordinate, effectively allowing the alignment of the lower-left origins of the RT and DepthBuffer. [Note that the previous DBCOD feature performed an optional translation of post-viewport-mapping RT pixel (screen) coordinates to generate DepthBuffer pixel (window) coordinates. Specifically, the Draw Rect Origin X,Y state could be subtracted from the RT pixel coordinates.]

This function uses **Depth Coordinate Offset X,Y** state (signed 16-bit values in 3DSTATE_DEPTH_RECTANGLE) that is unconditionally added to the RT pixel coordinates to generate DepthBuffer pixel coordinates.

The previous DBCOB feature can be supported by having the driver program Depth Coordinate X,Y Offset to the two's complement of the Draw Rect Origin. By programming Depth Coordinate X,Y Offset to zeros, the current "normal" operation (DBCOD disabled) can be achieved.

Programming Note	
Context:	Depth Coordinate Offset
• Only simple 2D RTs are supported (no mipmaps). • Software must ensure that the resultant DepthBuffer Coordinate X,Y values are non-negative. • There are alignment restrictions - see 3DSTATE_DEPTH_BUFFER command.on SFID_DP_DC2) are IA coherent.	

Stencil Test

The Stencil Test function can be used to discard pixels based on a comparison between the [**Backface**] **Stencil Test Reference** state variable and the pixel's stencil value. This is a general-purpose function used for such effects as shadow volumes, per-pixel clipping, etc. The result of this comparison is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Stencil Test Enable** state variable. If ENABLED, the current stencil buffer value for this pixel is read.

Programming Note	
Context:	Color Calculator - Stencil Test
If the Depth Buffer is either undefined or does not have a surface format of D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT and separate stencil buffer is disabled, **Stencil Test Enable** must be DISABLED.	

A 2nd set of the stencil test state variables is provided so that pixels from back-facing objects, assuming they are not culled, can have a stencil test performed on them separate from the test for normal front-facing objects. The separate stencil test for back-facing objects can be enabled via the **Double Sided Stencil Enable** state variable. Otherwise, non-culled back-facing objects will use the same test function, mask and reference value as front-facing objects. The 2nd stencil state for back-facing objects is most commonly used to improve the performance of rendering shadow volumes which require a different stencil buffer operation depending on whether pixels rendered are from a front-facing or back-facing object. The backface stencil state removes the requirement to render the shadow volumes in 2 passes or sort the objects into front-facing and back-facing lists.

The remainder of this subsection describes the function in term of [**Backface**] <**state variable name**>. The Backface set of state variables are only used if Double Sided Stencil Enable is ENABLED and the object is considered back-facing. Otherwise, the normal (front-facing) state variables are used.

This function then compares the [**Backface**] **Stencil Test Reference** value and the pixel's stencil value value after logically ANDing both values by [**Backface**] **Stencil Test Mask**. The comparison performed is

specified by the [**Backface**] **Stencil Test Function** state variable. The result of the comparison is passed down the pipeline for use in the Stencil Buffer Update function. The Stencil Test function does not in itself discard pixels.

If **Stencil Test Enable** is DISABLED, a result of "stencil test passed" is propagated down the pipeline.

Depth Test

The Depth Test function can be used to discard pixels based on a comparison between the incoming pixel's depth value and the current depth buffer value associated with the pixel. This function is typically used to perform the "Z Buffer" hidden surface removal. The result of this pipeline function is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Depth Test Enable** state variable. If enabled, the pixel's ("source") depth value is first computed. After computation the pixel's depth value is clamped to the range defined by **Minimum Depth** and **Maximum Depth** in the selected CC_VIEWPORT state. Then the current ("destination") depth buffer value for this pixel is read.

This function then compares the source and destination depth values. The comparison performed is specified by the **Depth Test Function** state variable.

The result of the comparison is propagated down the pipeline for use in the subsequent Depth Buffer Update function. The Depth Test function does not in itself discard pixels.

If **Depth Test Enable** is DISABLED, a result of "depth test passed" is propagated down the pipeline.

Programming Note:

- Enabling the Depth Test function without defining a Depth Buffer is UNDEFINED.

Pre-Blend Color Clamping

Pre-Blend Color Clamping, controlled via **Pre-Blend Color Clamp Enable** OR **Pre-Blend Source Only Clamp Enable** and **Color Clamp Range** states in COLOR_CALC_STATE, is affected by the enabling of Color Buffer Blend as described below.

The following table summarizes the requirements involved with Pre-/Post-Blend Color Clamping.

Programming Note	
Context:	Negative Values
Errata - Negative values on Unsigned Float channels are always clamped to 0 if blending is enabled, regardless of how pre-blend clamping is programmed.	

Programming Note	
Context:	Errata
If the following conditions are true, blend is optimized to a fill operation in hardware which may cause corruption to occur if the pixel value is out of the RT format range. This is due to both pre-blend and post-blend clamping being skipped under these conditions which results in the unclamped value not getting down converted to RT format. RT format = r10g10b10_float_a2_unorm Blend State.Blend Enable = true Pre-Blend clamp enable = 0 Source blend factor is programmed such that it resolves to 1.0f and Destination blend factor is programmed such that it resolves to 0.0f (blend is converted to fill)	

Blending	Pre-Blend Color Clamp	Clamp Range
Off	Disabled: clamp to RT range (1)	Must set range = RT range (except if Pre Blend Source Only Clamp Enable is set)
	Enabled: clamp to RT range (1)	Must set range = RT range
On (if permitted)	Disabled: clamp to internal format (1) for float RTs and to RT range for UNORM/SNORM RTs	Must set range = RT range
	Enabled: clamp to RT range (1)	Must set range = RT range

(1) If Pre Blend Source Only Clamp Enable is set in BLEND STATE, SourceColor is clamped to COLORCLAMP_UNORM

Values written to a render target are always clamped to the RT range.

Pre-Blend Color Clamping When Blending is Disabled

The clamping of source color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all source color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed.

Programming Note	
Context:	Pre-Blend Color Clamping When Blending is Disabled

- Given the possibility of writing UNPREDICTABLE values to the Color Buffer, it is expected and highly recommended that, when blending is disabled, software set **Pre-Blend Color Clamp Enable** to ENABLED and select an appropriate **Color Clamp Range**.
- When using SINT or UINT rendertarget surface formats, **Blending must** be DISABLED. The **Pre-Blend Color Clamp Enable** and **Color Clamp Range** fields are ignored, and an implied clamp to the rendertarget surface format is performed.

Pre-Blend Color Clamping When Blending is Enabled

The clamping of source, destination and constant color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all these color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed on these color components prior to blending.

Color Buffer Blending

The Color Buffer Blending function is used to combine one or two incoming "source" pixel color+alpha values with the "destination" color+alpha read from the corresponding location in a RenderTarget.

Blending is enabled on a global basis by the **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE). If DISABLED, Blending and Post-Blend Clamp functions are disabled for all RenderTargets, and the pixel values (possibly subject to Pre-Blend Clamp) are passed through unchanged.

The Color Buffer Blend Enable is in the per-render-target BLEND_STATE, and the field in SURFACE_STATE is no longer supported.

Programming Note	
Context:	Color Buffer Blending, Logic Ops, DataPort, surface formats, render targets

- Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
- Dual source blending: The DataPort only supports dual source blending with a SIMD8-style message.
- Only certain surface formats support Color Buffer Blending. Refer to the Surface Format tables in *Sampling Engine*. Blending must be disabled on a RenderTarget if blending is not supported.

The incoming "source" pixel values are modulated by a selected "source" blend factor, and the possibly gamma-decorrected "destination" values are modulated by a "destination" blend factor. These terms are then combined with a "blend function". In general:

src_term = src_blend_factor * src_color

dst_term = dst_blend_factor * dst_color

color output = blend_function(src_term, dst_term)

If there is no alpha value contained in the Color Buffer, a default value of 1.0 is used and, correspondingly, there is no alpha component computed by this function.

Dual Source Blending: When using "Dual Source" Render Target Write messages, the Source1 pixel color+alpha passed in the message can be selected as a src/dst blend factor (see "Color Buffer Blend Color Factors"). In single-source mode, those blend factor selections are invalid. If SRC1 is included in a src/dst blend factor and a DualSource RT Write message is not used, results are UNDEFINED. (This reflects the same restriction in DX APIs, where undefined results are produced if "o1" is not written by a PS - there are no default values defined). If SRC1 is not included in a src/dst blend factor, dual source blending must be disabled.

The blending of the color and alpha components is controlled with two separate (color and alpha) sets of state variables. However, if the **Independent Alpha Blend Enable** state variable in COLOR_CALC_STATE is DISABLED, then the "color" (rather than "alpha") set of state variables is used for both color and alpha. Note that this is the only use of the **Independent Alpha Blend Enable** state - it does not control whether Blending occurs, only how.

Per **Render Target Blend State**: Blend state is selected based on **Render Target Index** contained in the message header, and appropriate blend state is applied to Render Target Write messages.

The following table describes the color source and destination blend factors controlled by the **Source [Alpha] Blend Factor** and **Destination [Alpha] Blend Factor** state variables in COLOR_CALC_STATE. Note that the blend factors applied to the R,G,B channels are always controlled by the **Source/Destination Blend Factor**, while the blend factor applied to the alpha channel is controlled either by **Source/Destination Blend Factor** or **Source/Destination Alpha Blend Factor**.

Color Buffer Blend Color Factors

Blend Factor Selection	Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2nd output from PS in Dual-Souce mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_ZERO	0.0, 0.0, 0.0, 0.0
BLENDFACTOR_ONE	1.0, 1.0, 1.0, 1.0
BLENDFACTOR_SRC_COLOR	oN.r, oN.g, oN.b, oN.a
BLENDFACTOR_INV_SRC_COLOR	1.0-oN.r, 1.0-oN.g, 1.0-oN.b, 1.0-oN.a
BLENDFACTOR_SRC_ALPHA	oN.a, oN.a, oN.a, oN.a
BLENDFACTOR_INV_SRC_ALPHA	1.0-oN.a, 1.0-oN.a, 1.0-oN.a, 1.0-oN.a
BLENDFACTOR_SRC1_COLOR	o1.r, o1.g, o1.b, o1.a
BLENDFACTOR_INV_SRC1_COLOR	1.0-o1.r, 1.0-o1.g, 1.0-o1.b, 1.0-o1.a
BLENDFACTOR_SRC1_ALPHA	o1.a, o1.a, o1.a, o1.a

Blend Factor Selection	Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2nd output from PS in Dual-Souce mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_INV_SRC1_ALPHA	1.0-o1.a, 1.0-o1.a, 1.0-o1.a, 1.0-o1.a
BLENDFACTOR_DST_COLOR	rtN.r, rtN.g, rtN.b, rtN.a
BLENDFACTOR_INV_DST_COLOR	1.0-rtN.r, 1.0-rtN.g, 1.0-rtN.b, 1.0-rtN.a
BLENDFACTOR_DST_ALPHA	rtN.a, rtN.a, rtN.a, rtN.a
BLENDFACTOR_INV_DST_ALPHA	1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a
BLENDFACTOR_CONST_COLOR	CC.r, CC.g, CC.b, CC.a
BLENDFACTOR_INV_CONST_COLOR	1.0-CC.r, 1.0-CC.g, 1.0-CC.b, 1.0-CC.a
BLENDFACTOR_CONST_ALPHA	CC.a, CC.a, CC.a, CC.a
BLENDFACTOR_INV_CONST_ALPHA	1.0-CC.a, 1.0-CC.a, 1.0-CC.a, 1.0-CC.a
BLENDFACTOR_SRC_ALPHA_SATURATE	f,f,f,1.0 where f = min(1.0 - rtN.a, oN.a)

The following table lists the supported blending operations defined by the **Color Blend Function** state variable and the **Alpha Blend Function** state variable (when in independent alpha blend mode).

Color Buffer Blend Functions

Blend Function	Operation (for each color component)
BLENDFUNCTION_ADD	SrcColor*SrcFactor + DstColor*DstFactor
BLENDFUNCTION_SUBTRACT	SrcColor*SrcFactor - DstColor*DstFactor
BLENDFUNCTION_REVERSE_SUBTRACT	DstColor*DstFactor - SrcColor*SrcFactor
BLENDFUNCTION_MIN	min (SrcColor*SrcFactor, DstColor*DstFactor) **Programming Note:** This is a superset of the OpenGL "min" function.
BLENDFUNCTION_MAX	max (SrcColor*SrcFactor, DstColor*DstFactor) **Programming Note:** This is a superset of the OpenGL "max" function.

Post-Blend Color Clamping

(See *Pre-Blend Color Clamping* above for a summary table regarding clamping)

Post-Blend Color clamping is available only if Blending is enabled.

If Blending is enabled, the clamping of blending output color components is controlled by **Post-Blend Color Clamp Enable**. If ENABLED, the color components output from blending are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed at this point.

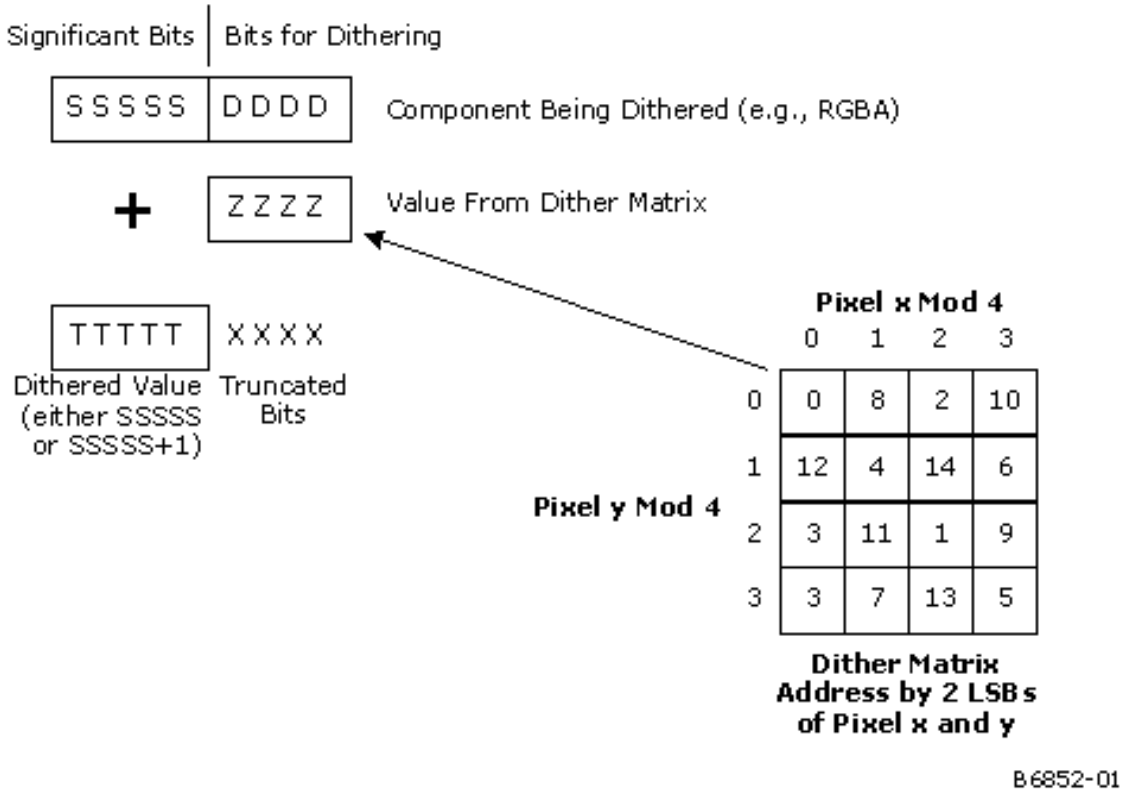
Regardless of the setting of **Post-Blend Color Clamp Enable**, when Blending is enabled color components will be automatically clamped to (at least) the rendertarget surface format range at this stage of the pipeline.

Dithering

Dithering is used to give the illusion of a higher resolution when using low-bpp channels in color buffers (e.g., with 16bpp color buffer). By carefully choosing an arrangement of lower resolution colors, colors otherwise not representable can be approximated, especially when seen at a distance where the viewer's eyes will average adjacent pixel colors. Color dithering tends to diffuse the sharp color bands seen on smooth-shaded objects.

A four-bit dither value is obtained from a 4x4 Dither Constant matrix depending on the pixel's X and Y screen coordinate. The pixel's X and Y screen coordinates are first offset by the **Dither Offset X** and **Dither Offset Y** state variables (these offsets are used to provide window-relative dithering). Then the two LSBs of the pixel's screen X coordinate are used to address a column in the dither matrix, and the two LSBs of the pixel's screen Y coordinate are used to address a row. This way, the matrix repeats every four pixels in both directions.

The value obtained is appropriately shifted to align with (what would be otherwise) truncated bits of the component being dithered. It is then added with the component and the result is truncated to the bit depth of the component given the color buffer format.



Dithering Process (5-Bit Example)

Logic Ops

The Logic Ops function is used to combine the incoming "source" pixel color/alpha values with the corresponding "destination" color/alpha contained in the ColorBuffer, using a logic function.

The Logic Op function is enabled by the **LogicOp Enable** state variable. If DISABLED, this function is ignored, and the incoming pixel values are passed through unchanged.

Programming Notes

Programming Note
Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
Logic Ops are supported on all blendable render targets and render targets with *INT formats.

The following table lists the supported logic ops. The logic op is selected using the **Logic Op Function** field in COLOR_CALC_STATE.

Logic Ops

LogicOp Function	Definition (S=Source, D=Destination)
LOGICOP_CLEAR	all 0's
LOGICOP_NOR	NOT (S OR D)
LOGICOP_AND_INVERTED	(NOT S) AND D
LOGICOP_COPY_INVERTED	NOT S
LOGICOP_AND_REVERSE	S AND NOT D
LOGICOP_INVERT	NOT D
LOGICOP_XOR	S XOR D
LOGICOP_NAND	NOT (S AND D)
LOGICOP_AND	S AND D
LOGICOP_EQUIV	NOT (S XOR D)
LOGICOP_NOOP	D
LOGICOP_OR_INVERTED	(NOT S) OR D
LOGICOP_COPY	S
LOGICOP_OR_REVERSE	S OR NOT D
LOGICOP_OR	S OR D
LOGICOP_SET	all 1's

Buffer Update

The Buffer Update function is responsible for updating the pixel's Stencil, Depth and Color Buffer contents based upon the results of the Stencil and Depth Test functions. Note that Kill Pixel and/or Alpha Test functions may have already discarded the pixel by this point.

Stencil Buffer Updates

If and only if stencil testing is enabled, the Stencil Buffer is updated according to the **Stencil Fail Op**, **Stencil Pass Depth Fail Op**, and **Stencil Pass Depth Pass Op** state (or their backface counterparts if **Double Sided Stencil Enable** is ENABLED and the pixel is from a back-facing object) and the results of the Stencil Test and Depth Test functions.

Stencil Fail Op and **Backface Stencil Fail Op** specify how/if the stencil buffer is modified if the stencil test fails. **Stencil Pass Depth Fail Op** and **Backface Stencil Pass Depth Fail Op** specify how/if the stencil buffer is modified if the stencil test passes but the depth test fails. **Stencil Pass Depth Pass Op** and **Backface Stencil Pass Depth Pass Op** specify how/if the stencil buffer is modified if both the stencil and depth tests pass. The operations (on the stencil buffer) that are to be performed under one of these (mutually exclusive) conditions is summarized in the following table.

Stencil Buffer Operations

Stencil Operation	Description
STENCILOP_KEEP	Do not modify the stencil buffer
STENCILOP_ZERO	Store a 0
STENCILOP_REPLACE	Store the *StencilTestReference* reference value
STENCILOP_INCRSAT	Saturating increment (clamp to max value)
STENCILOP_DECRSAT	Saturating decrement (clamp to 0)
STENCILOP_INCR	Increment (possible wraparound to 0)
STENCILOP_DECR	Decrement (possible wrap to max value)
STENCILOP_INVERT	Logically invert the stencil value

Any and all writes to the stencil portion of the depth buffer are enabled by the **Stencil Buffer Write Enable** state variable.

When writes are enabled, the **Stencil Buffer Write Mask** and **Backface Stencil Buffer Write Mask** state variables provide an 8-bit mask that selects which bits of the stencil write value are modified. Masked-off bits (i.e., mask bit == 0) are left unmodified in the Stencil Buffer.

Programming Note	
Context:	Stencil Buffer Updates
The Stencil Buffer can be written even if depth buffer writes are disabled via Depth Buffer Write Enable	

Depth Buffer Updates

Any and all writes to the Depth Buffer are enabled by the **Depth Buffer Write Enable** state variable. If there is no Depth Buffer, writes must be explicitly disabled with this state variable, or operation is UNDEFINED.

If depth testing is disabled or the depth test passed, the incoming pixel's depth value is written to the Depth Buffer. If depth testing is enabled and the depth test failed, the pixel is discarded - with no modification to the Depth or Color Buffers (though the Stencil Buffer may have been modified).

Color Gamma Correction

Computed RGB (not A) channels can be gamma-corrected prior to update of the Color Buffer.

This function is automatically invoked whenever the destination surface (render target) has an SRGB format (see surface formats in *Sampling Engine*). For these surfaces, the computed RGB values are converted from gamma=1.0 space to gamma=2.4 space by applying a ^(2.4) exponential function.

Color Buffer Updates

Finally, if the pixel has not been discarded by this point, the incoming pixel color is written into the Color Buffer. The **Surface Format** of the color buffer indicates which channel(s) are written (e.g., R8G8_UNORM are written with the Red and Green channels only). The **Color Buffer Component Write Disables** from the Color Buffer's SURFACE_STATE provide an independent write disable for each channel of the Color Buffer.

Pixel Pipeline State Summary

COLOR_CALC_STATE

3DSTATE_BLEND_STATE_POINTERS

3DSTATE_BLEND_STATE_POINTERS

3DSTATE_DEPTH_STENCIL_STATE_POINTERS

3DSTATE_DEPTH_STENCIL_STATE_POINTERS has been replaced by 3DSTATE_WM_DEPTH_STENCIL. (See Vol2a.11 3D Pipeline - Windower for details.)

COLOR_CALC_STATE

COLOR_CALC_STATE

DEPTH_STENCIL_STATE

DEPTH_STENCIL_STATE has been replaced by 3DSTATE_WM_DEPTH_STENCIL. (See *3D Pipeline - Windower* for details).

BLEND_STATE

BLEND_STATE

Signal	CC_INT::AlphaTestEnable
Description	AlphaTestEnable
Formula	= BLEND_STATE::AlphaTestEnable && !3DSTATE_WM_HZ_OP::DepthBufferResolveEnable && !3DSTATE_WM_HZ_OP::DepthBufferClear && !3DSTATE_WM_HZ_OP::StencilBufferClear

Signal	CC_INT::AlphaToCoverageEnable
Description	AlphaToCoverageEnable
Formula	= BLEND_STATE::AlphaToCoverageEnable && !3DSTATE_PS_EXTRA::PixelShaderDisableAlphaToCoverage

CC_VIEWPORT

CC_VIEWPORT

Other Pixel Pipeline Functions

Statistics Gathering

If **Statistics Enable** is set in 3DSTATE_WM, the PS_DEPTH_COUNT register (see Memory Interface Registers in Volume 1a, *GPU Overview*) is incremented once for each pixel (or sample) that passes the depth, stencil and alpha tests. Note that each of these tests is treated as passing if disabled. This count is accurate regardless of whether **Early Depth Test Enable** is set. To obtain the value from this register at a deterministic place in the primitive stream without flushing the pipeline, however, the PIPE_CONTROL command must be used. See Volume 2a, *3D Pipeline*, for details on PIPE_CONTROL.

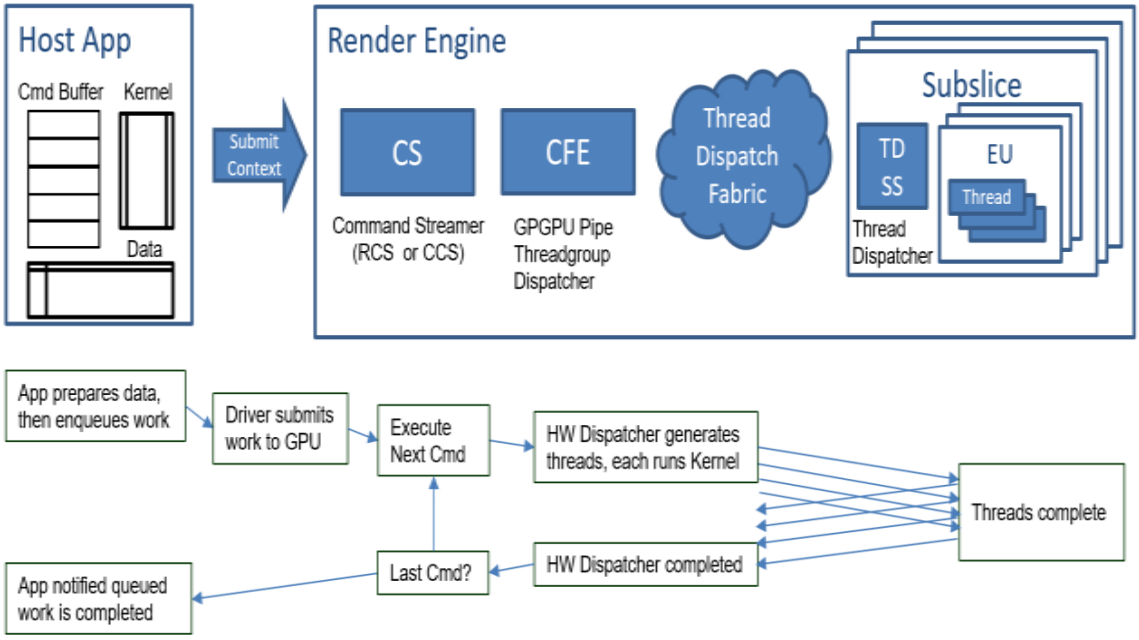
GPGPU Compute Pipeline

General Purpose GPU (GPGPU) applications use the GPU to perform highly parallel work on memory-based data. The work is performed by independent threads that are started by a hardware thread dispatcher, each running on one of many Execution Units in the GPU. When threads finish, their completions trigger either more threads to be dispatched or a completion to be sent back to the host application.

This chapter describes the execution environment, the command programming, and the thread synchronization mechanisms in the GPGPU Pipeline.

GPGPU Pipeline Overview

General purpose GPU (GPGPU) applications are a combination of CPU host software and GPU kernels. The host builds a command buffer to submit the GPU kernels for execution. The host submits the command buffer to either a Render Command Streamer or a Compute Command Streamer. The kernels run in parallel, usually independently of each other, reading and writing GPU memory. The host is notified when the kernels complete execution. The host software sets up the GPU memory prior to the running of the kernels and copies the GPU memory results back to the host after the kernels have completed.



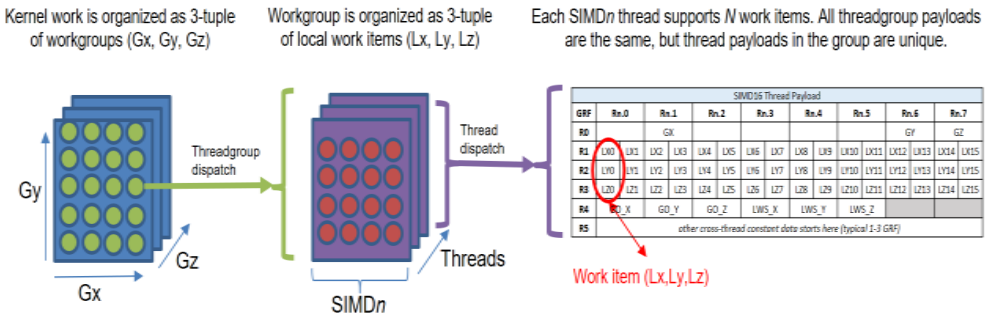
The command buffer is the top level of the GPGPU pipeline. The commands set up the pipeline state, dispatch work through kernels, and wait for completions. The sequence of commands determine which kernels can be started and run in parallel. Multiple independent kernels can be running at the same time. When there are order dependencies between different kernels, a wait command is used between their dispatch commands.

GPGPU Thread Model

The work performed by a kernel is executed in parallel by multiple threads running the same kernel, all using the same global parameters and a per-thread index to select the thread's portion of the global memory data. A single COMPUTE_WALKER command generates all the thread dispatches and thread parameters for a kernel to perform its work. See diagram below.

A kernel's work is divided into logically independent workgroups. Each workgroup is a batch of work items, which optionally can coordinate their execution through barriers and shared local memory. A workgroup is guaranteed to execute all its work items at the same time, but not in any specific order. There is no guarantee that multiple workgroups are running at the same time or in any specific order.

A threadgroup is the batch of threads that together execute all the workgroup's work items. Each thread executes multiple work items, usually one work item for each of the thread's SIMD lanes. Each threadgroup is dispatched to any one of the Subslice partitions in the GPU. The Subslice implements the barriers and shared local memory a threadgroup to optionally coordinate its threads' execution.



Kernel work is 3-tuple (X, Y, Z) of work items. Total work size = dim(X) * dim(Y) * dim(Z).
Work items are batched into workgroups. Local work size LWS = LWS_X * LWS_Y * LWS_Z.
Global work item (X, Y, Z) = (Gx*LWS_X + Lx, Gy*LWS_Y + Ly, Gz*LWS_Z + Lz).

GPGPU Memory Model

Different memory spaces are used by GPGPU kernels: global memory, threadgroup shared local memory, and thread private memory (Scratch Space). Each of these memory spaces has a hierarchy of accessibility and visibility:

Private memory	Accessible and visible to work items within a thread, but not to work items outside of that thread and not after the thread has terminated.
Shared local memory (SLM)	Accessible and visible to work items within a threadgroup, but not to work items outside of its threadgroup and not after the threadgroup has terminated.
Global memory	Accessible and visible to all work items within a context.
Shared virtual global memory (SVM)	Accessible and visible to all work items on any GPU and the host.

GPGPU reads and writes are typically executed with relaxed ordering. Memory writes are posted operations and may not be visible until after a memory fence release operation. Memory reads are cached and may not be consistent until after a memory fence acquire operation. Memory read-after-write and write-after-read access ordering is guaranteed either by the GPGPU application with memory fence release and acquire operations, or with careful ordering of accesses made within a single thread:

Different threads load/store	Any read and write accesses, by different threads in a threadgroup or across different threadgroups, require a memory fence operation to guarantee ordering. A thread synchronization operation (e.g., barrier or atomic semaphore) is required to guarantee that all threads have completed their memory fence ordering operation.

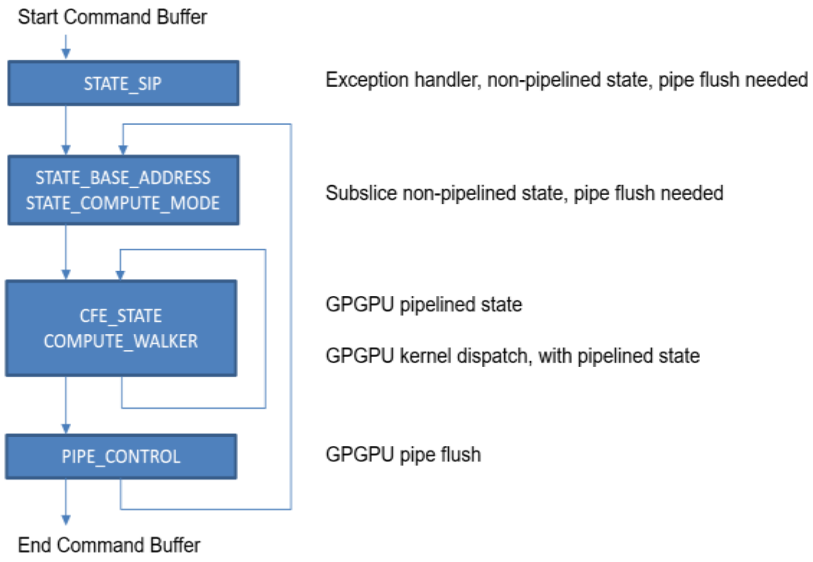
Same thread load/store	Read and write accesses made within the same thread are ordered only when made to the same address and the same address space. Accesses to different addresses within the same thread are unordered and may require a memory fence to guarantee ordering. A memory fence operation within a single thread orders all the memory accesses by the single thread.
Atomic operations by any thread	Atomic read-modify-write operations to the same address are always ordered with other atomic operations and do not require fences. However, atomic operations are unordered when used in combination with load or store operations to the same address.

The table below summarizes where explicit fence operations are required based on the class of memory accesses made by GPGPU kernels.

Memory Class		**Fence Required for each Memory Scope**			
GPGPU Operation	**Access**	**Thread**	**Threadgroup**	**Context Global**	**Host SVM**
Private Memory Load/Store	RW	None	*N/A*	*N/A*	*N/A*
Shared Local Memory Load/Store	RW	None	Release/acquire fence	*N/A*	*N/A*
Shared Local Memory Atomics	RW	None	None	*N/A*	*N/A*
Global Constants Load	RO	None	None	None	Host driver fence
Global Memory Load/Store	RW	None	Release/acquire fence	Release/acquire fence	Host driver fence
Global Memory Atomics	RW	None	None	None	Host driver fence
SVM Global Memory Load/Store	RW	None	None	None	None

Programming the GPGPU Pipeline

The GPGPU pipeline is programmed using commands that set up the state, dispatch kernels, and wait for completions. The typical sequence of commands is shown in the diagram below. State commands can be issued in any order. Issuing a state command is not required when its values are unchanged. The COMPUTE_WALKER command is used to dispatch kernels. The completion of dispatched kernels is tracked and handled by both COMPUTE_WALKER and PIPE_CONTROL commands.



Compute State Commands

The state commands set up parameters to control behavior of GPGPU threads and their memory operations. These parameters are typically set up at the beginning of GPGPU application context, and rarely change between submissions of kernels for execution.

Type	Command	Description
NP	**STATE_BASE_ADDRESS**	Base addresses for the Instruction, General State, Surface State, and Bindless Surface State memory heaps. GPGPU kernels reference these memory areas using 32b offsets from the 64b base addresses.
NP	**STATE_COMPUTE_MODE**	Mode settings for miscellaneous memory scope controls, including Subslice caches, coherency, fences, and atomic operations.
P	**CFE_STATE**	Thread dispatch state, including Scratch Space for thread private memory, and maximum active threads to dispatch.
NP	**STATE_SIP**	Address of function to call if a thread throws an exception during its execution. Usually, exceptions are ignored and the thread continues execution.

P	**PIPE_CONTROL**	Synchronize pipeline by waiting for all threads to complete, and optionally flush GPU caches. A subset of PIPE_CONTROL functions are also directly available in the COMPUTE_WALKER command.

The non-pipelined (NP) state commands require that no GPGPU threads are active prior to use. Typically, a PIPE_CONTROL command is inserted to wait until all the GPGPU threads have completed before the NP state command is issued. The PIPE_CONTROL command is not needed when a NP state command is issued at the start of a context submission before the first COMPUTE_WALKER command.

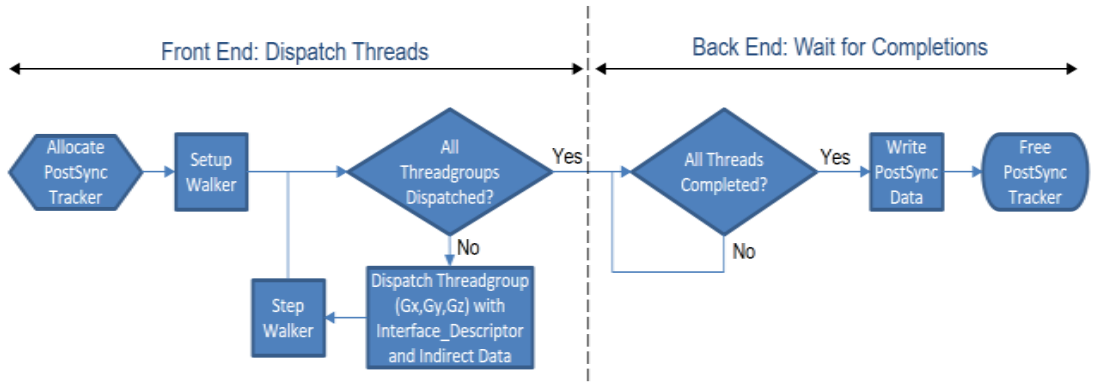
Compute Walker Command

The COMPUTE_WALKER command is used to dispatch all the parallel thread instances needed to execute a single GPGPU kernel over its full range of data. The compute walker dispatches threadgroups to every available Subslice in the GPU. Threadgroup dispatches are dynamically load balanced: each threadgroup dispatch is sent to the least loaded Subslice.

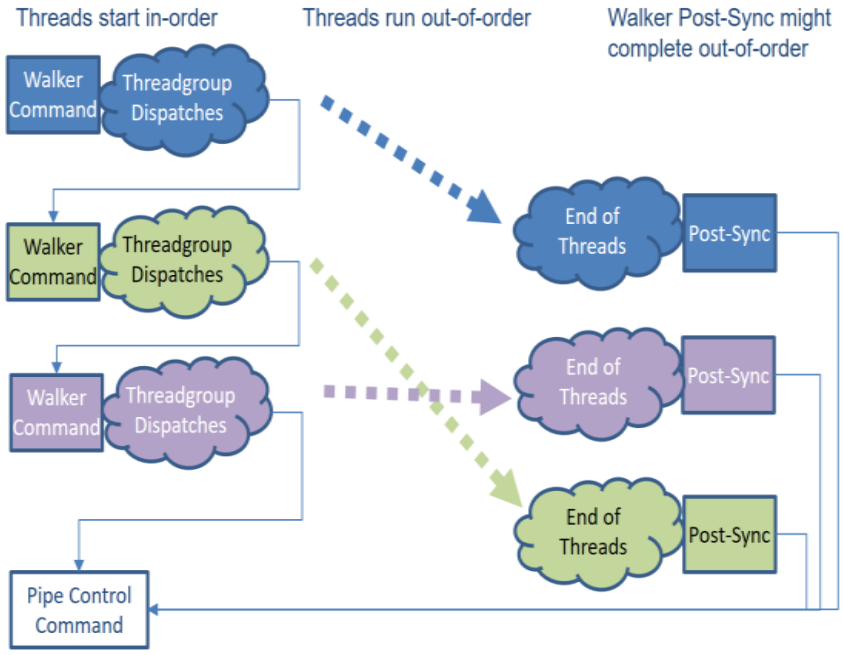
Parameters to the **COMPUTE_WALKER** command include:

Interface Descriptor Data	Kernel start address, number of threads per threadgroup, SLM and barrier allocations
X/Y/Z Dimensions	Range of threadgroups to dispatch, each as (Gx,Gy,Gz)
Indirect Data Address/Length	Location of per-thread and cross-thread parameters for each thread in a threadgroup dispatch
PostSync Descriptor Data	Memory address to update when all threads dispatched by this walker command have completed

The COMPUTE_WALKER command has two independent phases: a front end that dispatches threadgroups, and a back end that executes the PostSync operations when the threads have finished running. See diagram below.



Threads dispatched from different walker commands can be running concurrently. Separating the in-order dispatching of threads from sequential walker commands and the out-of-order completion of those threads for Post Sync operations combines an “oldest-first” thread dispatch policy that fills all the available resources, with completion tracking that does not drain the dispatch pipeline. See diagram below.



Pipeline Synchronization Operations

The synchronization of the GPGPU pipeline is required when the operations from a previous set of threads needs to be finished before starting the operations of the next set of threads. The ordering around the synchronization point includes both a fence operation for thread dispatch order, and a memory fence operation for the memory access order.

Synchronization Command	Thread Fence Description	Memory Fence Control
PIPE_CONTROL	Acts as barrier operation in the command stream. Command stream waits until all threads are completed before continuing to the next command.	After the threads are completed, optional SS_FLUSH and L3_FLUSH operations are performed.
SEMAPHORE_WAIT on PostSync	Command stream waits until the memory location has been written by a COMPUTE_WALKER PostSync operation.	SS_FLUSH is performed before the PostSync operation updates the memory location.

The Dataport Subslice Cache Flush (SS_FLUSH) operation is a memory release fence followed by a memory acquire fence for all the Subslice partitions in the GPU. This ensures that all memory writes have been completed, and that the next memory read will pick up the globally visible value of the memory location (and not some Subslice cached version of the data).

The COMPUTE_WALKER PostSync operation has a Destination Address, an Immediate Data field, and some Control bits to select which type of operation to perform. See **POSTSYNC_DATA**. The driver or the SEMAPHORE_WAIT command will detect completion of walker thread when the memory location is written with a different value than it was initialized with.

PostSync Operation	Description
None	No write.
Write Immediate Data	Write 8 bytes of immediate data to the 64-bit aligned destination address.
Write Timestamp Packet	Write 16 bytes of timestamp payload to the 128-bit aligned destination address. The timestamp can be used for performance measurements.

Overdispatch

The Compute Pipeline rapidly dispatches threads to fill every available thread slot. The goal of the thread dispatcher is to have the next thread to dispatch waiting inside the subslice for when the next EU thread becomes available. While this minimizes the idle slots in the EU, it also causes thread overdispatch. Overdispatch is a temporary condition where more threads have been dispatched than can be concurrently run.

The CFE_STATE command supports two methods to control the overdispatch:

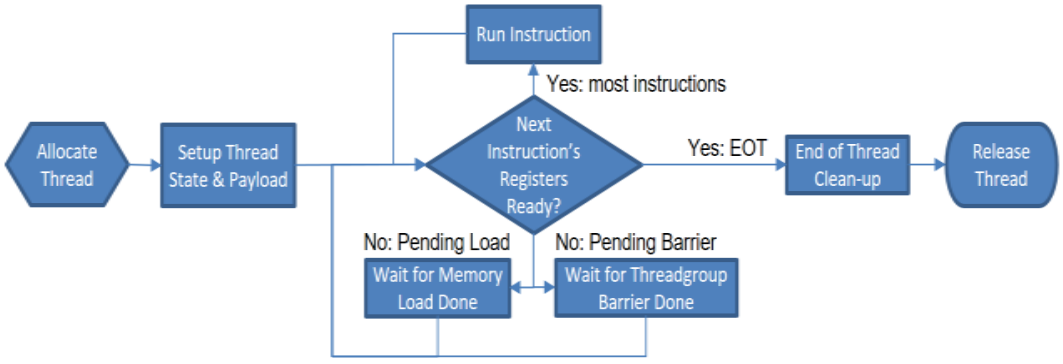
Parameter	Description	Example Use
Maximum Number of Threads	At the top of the Compute Pipeline, limit the number of active threads to the specified number.	When a few threads will fully use the memory bandwidth or compute bandwidth of the subslice.
Overdispatch Control	Within the various thread dispatch buffers, limit the number of dispatched threads.	When threads have short runtimes, increase the thread overdispatch to hide the latency of thread dispatch.

The thread dispatcher tries to dispatch a threadgroup to the least-loaded subslice. With the dynamic load-balancing of threads, sometimes the thread dispatcher temporarily overdispatches to one subslice and underdispatches to another subslice. When the number of thread dispatches is much larger than the number of threads in the GPU, this does not cause a performance imbalance. However, when the number of threads is a small multiple of the number of threads in the GPU, then a performance imbalance can occur.

Programming Note	
Context:	Perfect load-balanced thread dispatch
When perfect load balancing of thread dispatches to subslices is required, SW can achieve this by dispatching more threads than are required, and the extra threads exit without doing any work. For example, assume a program wants exactly 10 threads to run on every subslice. Then the COMPUTE_WALKER could be programmed to dispatch ((number of threads in the configuration) * (1 + overdispatch_factor)) + (1 extra thread per subslice). Then on each subslice, every thread uses an atomic counter to determine if it one of the 10 to run, and if not then exit.	

GPGPU Thread Dispatch and Execution

A thread runs on an Execution Unit (EU). Each EU holds the thread state for all of its active threads. The threads are either waiting, ready-to-run, or running. Only one thread is running on the EU at any time. When a thread is waiting, the EU switches execution to its next ready-to-run thread. When a thread exits, that thread slot becomes available for the next thread dispatch.



Thread Execution Mask

Each thread dispatch specifies an Execution Mask, one enable bit for each of the active SIMT lanes in each dispatched thread. The same kernel code can be dispatched and run with a different number of active lanes.

The Execution Mask is dynamically computed for each thread dispatched in a threadgroup. A partial execution mask is generated only on the last thread dispatched in a threadgroup. The last thread's Execution Mask is specified in the COMPUTE_WALKER command.

Thread Payloads

After setting up EU thread state at thread dispatch, the thread payload is copied sequentially into the thread's general register file starting at R0. The format of R0 is the same for all GPGPU threads. After R0,

additional thread payload registers may be generated based on the COMPUTE_WALKER payload controls. The host application driver decides which parameter passing convention to use with the GPU kernel and sets the COMPUTE_WALKER controls accordingly.

	Thread Payload Description
R0	All compute threads have a common parameter layout for their first register. See register layout in **R0**.
R1 Bindless Thread	Setting the Bindless Thread Mode in the COMPUTE_WALKER Interface Descriptor causes the HW to load R1 with the Bindless Thread stack IDs. See **BTD Mode R1** for register payload layout.
Local ID	Setting the Emit Local controls in the COMPUTE_WALKER cause the HW to generate register payloads for local work items X/Y/Z. See register layout in **Local ID X/Y/Z (R1..R3)**.
Inline	Setting the Emit Inline control in the COMPUTE_WALKER causes the HW to load the next register (after the local ID registers) with a constant value copied from the COMPUTE_WALKER command. See register layout in **Inline Data**. The remainder of the cross-thread payload is provided in Indirect Data payload. The Inline cross-thread payload for the thread is normally reserved to pass pointers and remove one level of fetch indirection from the kernel.
Indirect Data	If Indirect Data Length > 0 in the COMPUTE_WALKER, then the address of threadgroup payload is present in Indirect Data Address in R0. The Indirect Data payload is not loaded by HW into the thread's registers: the compiler inserts prolog code in the kernel to load the thread's payload into the registers.

Thread Groups

A GPGPU threadgroup dispatch within a Subslice partition launches a sequence of threads to the least-loaded Execution Units (EU) in that Subslice. Most of the thread state is the same and shared across all the threads in the threadgroup. Unique to each thread dispatch is the EU thread ID, the local work items in the thread payload, and the scratch space for the thread.

All threads within a threadgroup are guaranteed to be resident and runnable in the threadgroup's Subslice. Normally each thread within a threadgroup is dispatched to a different EU to maximize overlapped execution of the threadgroup. Sometimes the size of the threadgroup or the dynamic loading of the Subslice causes two or more threads in the threadgroup to be dispatched to the same EU. Dispatching threads to the same EU may result in lower performance but does not create a deadlock condition.

The shared resources of a threadgroup are dynamically allocated at the start of a threadgroup dispatch. These resources are released when the last thread in each threadgroup ends.

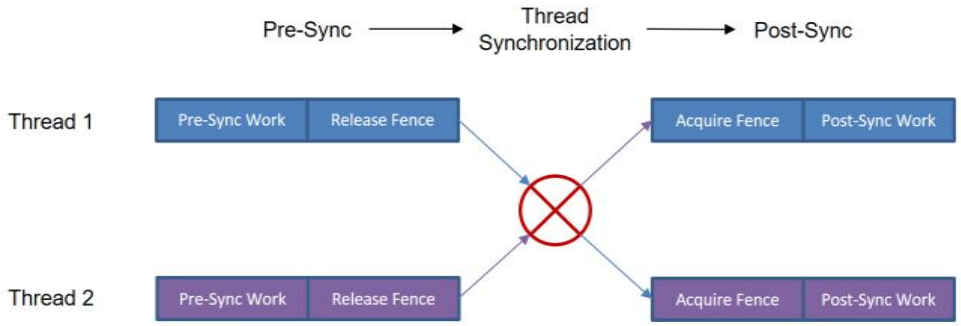
Resource	Description
Barriers	The interface descriptor in the COMPUTE_WALKER specifies if the threadgroup requires any barriers. If required, the threadgroup dispatch assigns the unique barrier IDs for each thread dispatch in the threadgroup.
Shared Local Memory (SLM)	The interface descriptor in the COMPUTE_WALKER specifies if the threadgroup requires SLM. If required, the threadgroup dispatch allocates the required amount of SLM and provides its unique address for each thread dispatch in the threadgroup.

Thread Synchronization

Thread synchronization is used in applications when pipeline level synchronization of threads is too coarse-grained for good performance.

Type	Description
Threadgroup Barrier	The barrier operation is used to synchronize the completion of the work items performed by the threads in a threadgroup, prior to starting the next phase or iteration of work by the same threadgroup. The barrier operation is efficient. The overhead of the barrier operation is less than the overhead of an atomic memory operation. Typically, all the threads in a threadgroup require about the same amount of time to complete their work, so the wait time for the barrier is only the small difference in time from the first to last thread reaching the barrier point.
Atomic Semaphore	Any two or more threads running on the GPU can synchronize their work using atomic operations on a shared memory location. This is useful for producer-consumer applications, or for a large pool of cooperating threads. However, deadlock is possible unless the application guarantees that all the participating threads are present in the GPU. Threads waiting to synchronize repeatedly perform an atomic read of the shared location until the synchronization condition is matched (for example, a counter reaching zero). This polling loop is less efficient than barriers because it increases memory accesses in the kernels. Thread Synchronization Event Monitors (insert link) can be used to reduce the overhead of atomic polling by having a thread wait until the synchronization event has been signaled. The event monitors work in combination with atomic memory operations, providing a notification mechanism to reduce the atomic polling overhead.

With thread level synchronization, global memory ordering is explicitly guaranteed with release and acquire fence operations inside the kernel. See diagram below. A per-thread release fence is used to wait for previously issued memory operations to be globally visible in the GPU before starting the synchronization operation. A per-thread acquire fence is used to remove copies of global memory data from local caches to ensure that the globally visible data is used after the synchronization operation.



End of Thread

The last EU instruction executed by a thread has the EOT instruction option bit set on a EU SEND instruction. A GPGPU thread exits by sending an EOT message to the SFID_GW shared function. Immediately after the EOT message has been sent, the EU releases the thread and makes it available for the next pending thread dispatch.

All output from the GPGPU thread is written to global memory prior to issuing the EOT message. However, those writes may not yet be globally observable after the thread exits. A release fence is required to guarantee memory ordering before reading that global memory, either by using HDC_FLUSH at the pipeline level, or by using Memory_Fence operation before the thread exits.

Bindless Threads

Before a GPGPU thread exits, it can optionally dispatch one or more Bindless Threads that will execute on the same Subslice. These dispatches are made by sending dispatch messages to the SFID_BTD shared function.

If a GPGPU thread will dispatch a bindless thread, a Bindless Thread Stack must be allocated. The interface descriptor in the COMPUTE_WALKER specifies if the GPGPU thread requires bindless thread stacks. If required, a separate stack is allocated for each active SIMT lane in the GPGPU thread (specified by the thread's execution mask).

A Bindless Thread can optionally dispatch additional Bindless Threads. Those threads will use the same stack. The bindless thread stack, allocated by the threadgroup dispatch, is deallocated only when the last bindless thread that uses that stack exits.

Bindless threads are not part of the threadgroup that that dispatched them, and do not have access to the threadgroup's barriers or shared local memory.

In addition to waiting for all the threadgroups to exit, the COMPUTE_WALKER PostSync operation is not triggered until the deallocation of all the bindless thread stacks allocated by this COMPUTE_WALKER's threadgroup dispatches.

GPGPU Context

The GPGPU Context is the memory and execution environment present when running GPGPU programs. The GPGPU host program submits work to the GPU through a Context Descriptor that points to the context image, the command buffer, and the PPGTT. The context image holds the HW state that is restored to the GPU prior to starting the work in the command buffer. The PPGTT is the page table that maps the GPU virtual addresses to their physical memory addresses. The GPU driver manages the PPGTT and the physical memory that is allocated to it. The command buffer contains the commands that will be run on the GPU.

See Render Engine Context Submission for more details on the context image.

There are two types of GPGPU context images: Render Command Stream and Compute Command Stream. The Render Command Stream supports both 3D and GPGPU pipelines, switching between them with the PIPE_SELECT command. The Compute Command Stream only supports the GPGPU pipeline. The context image for the Render Command Stream is larger and holds both GPGPU and 3D engine state. The two types of context images are not interchangeable, but the GPGPU environment is the same in either context.

Context Switch

When a GPGPU context is started, previously saved GPU state is read from the context memory image and written into the GPU internal state. Next, the GPGPU pipeline starts fetching the first command from the command buffer.

When a GPGPU context is finished, the context image is written with the current GPU state, the currently executing COMPUTE_WALKER command (if any), and the pointer in the command buffer to the next command to run. When re-submitted, this context image will continue execution using the last GPU state that was present from the previous commands.

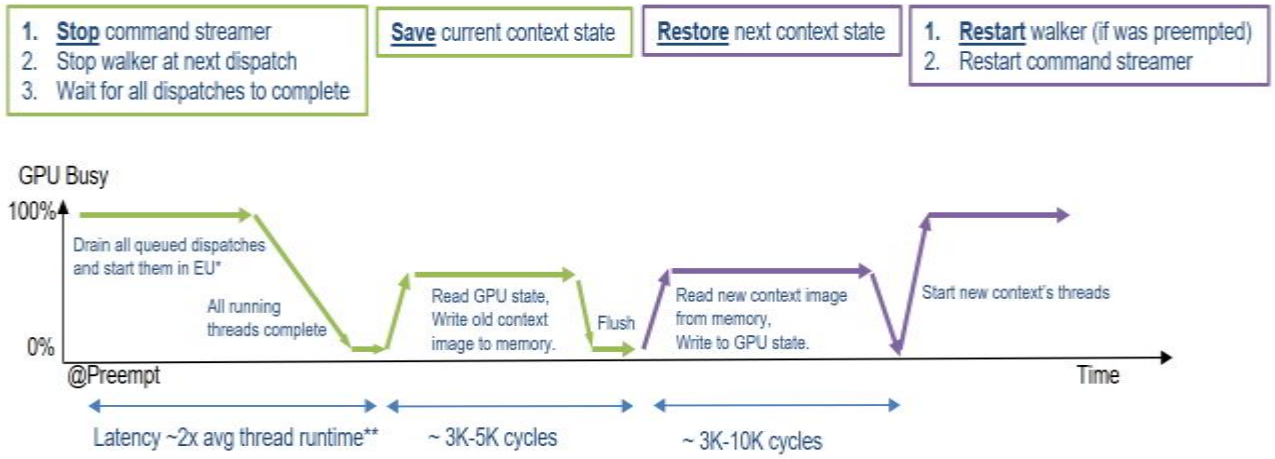
Preemption

A running GPGPU context can be preempted when the GPU driver submits a new context in place of the currently running GPGPU context. Typically, GPGPU contexts finish quickly enough that they can be run to completion without preemption and quickly enough to meet the system's expectations of GPU "quality of service". The GPU driver may choose to preempt a long-running GPGPU context if the running context has exceeded a time slice, or if a higher-priority submission is pending.

The GPGPU pipe supports 2 types of preemption boundaries: command and threadgroup. When preemption is signaled the GPGPU pipe stops execution at the next command preemption boundary and saves the GPU state into the context image.

Preemption Boundary	Description
Command	The context is preempted after the current command is completed. The GPGPU pipe is flushed, and the current context state is saved so that the next command in the command buffer will be executed when the context is resumed.
Threadgroup	The context is preempted at any command boundary, or in the middle of a COMPUTE_WALKER command. After all the previously dispatched threadgroups have completed running, the current walker position is saved along with the current context state. The current walker command will be executed when the context is resumed and then re-start at the next threadgroup dispatch in the walker.
Mid-thread	There is no hardware support to preempt a running thread and save the per thread state. **Programming Note:** GPGPU kernels that need mid-thread preemption will implement a software checkpoint scheme: checking a global variable to determine when they should save their intermediate work to memory and exit early, in combination with the host application detecting incomplete work and re-scheduling that work for completion.

The responsiveness of the GPU to preempt a GPGPU context depends on the whether a COMPUTE_WALKER is running and how long it takes for any running threads to complete. The diagram below shows what to expect for when going through the 4 phases of preemption.



** Why 2x? Queued threadgroup dispatches is subset of total GPU threads, so all queued threadgroups will start in less time than it takes one full thread to run to completion. However, when preempting at command level this time can be much longer if the COMPUTE_WALKER has many dispatches.

Multi-Context

Multiple Compute Command Streams (CCS) can be running simultaneously. In the simplest usage, one CCS is used by the entire GPU and its work is distributed across all the GPU subslices. When multiple CCS are used, each gets a separate partition of the GPU subslices, and executes on a separate GPGPU pipeline. The number of CCS available in the GPU is a configuration parameter for each hardware product.

Each CCS uses its own virtual address space (PPGTT). A common usage of multiple CCS is to support separate host applications running concurrently, instead of them being timesliced on the whole GPU. The concurrent execution allows some overlapped execution and removes some overhead with 4-step preemption process. See table, where the skipped steps are shown in gray.

Use Case	Preemption Steps			
Full preempt of CCS context with another CCS context	Stop	Save	Restore	Restart
Preempt CCS context, leaving GPU idle	Stop	Save		
Add CCS context while GPU is idle			Restore	Restart
Add CCS context while a different CCS context is running. Running CCS contexts are stopped on the subslices used by new CCS.	Stop		Restore	Restart
CCS context finishes while a different CCS context is running. Running CCS contexts take over the subslices released by the finished CCS.		Save		Restart

Another multi-context usage is the Render Command Stream (RCS) running simultaneously with one CCS. For example, a 3D application using Async Compute. Because these share the same virtual address space, they will also execute sharing the same subslices.

Multi-Partition

The partitioned COMPUTE_WALKER command enables the automatic distribution of threadgroup dispatches from a single COMPUTE_WALKER command to multiple independently executing command streams. By sub-selecting the walker dispatches in batches of the Partition Size parameter, the single walker's work is automatically divided across multiple engines for concurrent execution.

Parameters used for COMPUTE_WALKER partitions are:

Partition Type	Specifies whether the COMPUTE_WALKER is partitioned or not. If partitioned, specifies the single dimension (X, Y, or Z) to split.
Partition Size	Specifies how many threadgroups to generate in the partitioned dimension. All threadgroups are dispatched by this walker in the non-partitioned dimension.
Partition ID	Specifies which partition to generate in the partitioned dimension.
Workload Partition Enable	Uses WPARID MMIO register in place of the Partition ID for this COMPUTE_WALKER. The WPARID register is set in the command streamer prior to executing the command.

The driver must allocate the Scratch Space for all partitions as a contiguous block of memory. The COMPUTE_WALKER command adjusts the Scratch Space Base Address when the Partition ID > 0. The Scratch Space Base Address for a ParititionID is (CFE_STATE.ScratchSpaceBase + (PartitionID * CFE_STATE.MaxThreads * CFE_STATE.PerThreadScratchSize)). CFE_STATE.MaxThreads is programmed "per Partition".

Programming Note	
Context:	Multi-Partition Examples

To detect that the single walker's work is complete across all the partitions, additional command stream programming is added to the command stream.

Consider the case when the command stream is a sequence of 4 commands: (1) Walker1 , (2) Walker2, (3) PIPE_CONTROL barrier, (4) Walker3. The table below shows three different ways this could be partitioned for execution. The highlight text shows the differences in the command streams.

Partition Use Case	Command Stream Pseudo-Code
Single Partition	COMPUTE_WALKER(par_id=0, npar=1, &w1_postsync, … other walker1 params) COMPUTE_WALKER(par_id=0, npar=1, &w2_postsync, … other walker2 params) PIPE_CONTROL COMPUTE_WALKER(par_id=0, npar=1, &w3_postsync, … other walker3 params)

Fixed 4 Partitions	mypar_id = get_my_context_virtual_engine() // 0..3
	COMPUTE_WALKER(mypar_id, npar=4, &w1_postsync, … other walker1 params)
	COMPUTE_WALKER(mypar_id, npar=4, &w2_postsync, … other walker2 params)
	PIPE_CONTROL(L3_Flush) // my threads done, then make my L3 visible to other partitions
	for par_id=(0..3) { // wait for all partitions to be done
	MI_SEMAPHORE_WAIT(&w1_postsync[par_id])
	MI_SEMAPHORE_WAIT(&w2_postsync[par_id])
	}
	COMPUTE_WALKER(mypar_id, npar=4, &w3_postsync, … other walker3 params)

Load Balanced across N Partitions	while ((mypar_id = MI_ATOMIC_INC(&w1_tile)) < NPARITIONS) {
	COMPUTE_WALKER(mypar_id,npar=NPARTITIONS, &w1_postsync, … other walker1 params)
	}
	while ((mypar_id = MI_ATOMIC_INC(&w2_tile)) < NPARITIONS) {
	COMPUTE_WALKER(mypar_id, npar=NPARTITIONS, &w2_postsync, … other walker2 params)
	}
	PIPE_CONTROL(L3_Flush) // my threads done, then make my L3 visible to other partitions
	for par_id=(0..NPARTITIONS-1) { // wait for all partitions to be done
	MI_SEMAPHORE_WAIT(&w1_postsync[par_id])
	MI_SEMAPHORE_WAIT(&w2_postsync[par_id])
	}
	while ((mypar_id = MI_ATOMIC_INC(&w3_tile)) < NPARITIONS) {
	COMPUTE_WALKER(mypar_id, npar=NPARTITIONS, &w3_postsync, … other walker3 params)
	}

3D and GPGPU Programs

Overview of kernel execution with EU instructions and Shared Function messages.

EU Overview

The instruction set is a general-purpose data-parallel instruction set optimized for graphics and media computations. Support for 3D graphics API (Application Programming Interface) Shader instructions is mostly native, meaning that GFX efficiently executes Shader programs. Depending on Shader program operation modes (for example, a Vertex Shader may be executed on a base of a vertex pair, while a Pixel Shader may be executed on a base of a 16-pixel group), translation from 3D graphics API Shader instruction streams into native instructions may be required. In addition, there are many specific capabilities that accelerate media applications. The following feature list summarizes the instruction set architecture:

- SIMD (single instruction multiple data) instructions. The maximum number of data elements per instruction depends on the data type.
- SIMD parallel arithmetic, vector arithmetic, logical, and SIMD control/branch instructions.
- Instruction level variable-width SIMD execution.
- Conditional SIMD execution via destination mask, predication, and execution mask.
- Instruction compaction.
- An instruction may be executed in multiple cycles over a SIMD execution pipeline.
- Most instructions have three operands. Some instructions have additional implied source or destination operands. Some instructions have explicit dual destinations.
- Region-based register addressing.
- Direct or indirect (indexed) register addressing.
- Scalar or vector immediate source operand.
- Higher precision accumulator registers are architecturally visible.
- Self-modifying code is not allowed (instruction streams, including instruction caches, are read-only).

CoIssue/Multi Issue:

Multiple execution pipes operate simultaneously, scheduler selects an execution pipe based on their availability and capability. Each instruction may occupy multiple **execution slots** of the execution pipe based on their SIMD width, e.g., a SIMD16 instruction executed on a SIMD4 wide execution pipe will occupy 4 slots. Some instructions are defined as macros that expand into multiple elemental instructions inside the pipe occupying multiple execution slots. e.g., a dpas.8x8 instruction expands into 8 elemental dpas.8x1 instructions and will occupy 8 slots.

Thread scheduling:

Threads are scheduled with the "oldest first" policy: a thread runs as long as no dependency is encountered. When a switch is required, the oldest thread i.e., the thread which has been spawned the

first is the next to execute. After scheduling the next instruction from the currently executing thread, if any of the four units are free, the EU tries to fill them from instructions from other threads (processed in oldest to newest order).

A new "round-robin" scheduling policy is also introduced. This new policy issues the thread in a *round robin* fashion as opposed to *oldest first*. It is a global selection policy meaning that all the EUs are selected to run one policy or the other in an any given time.

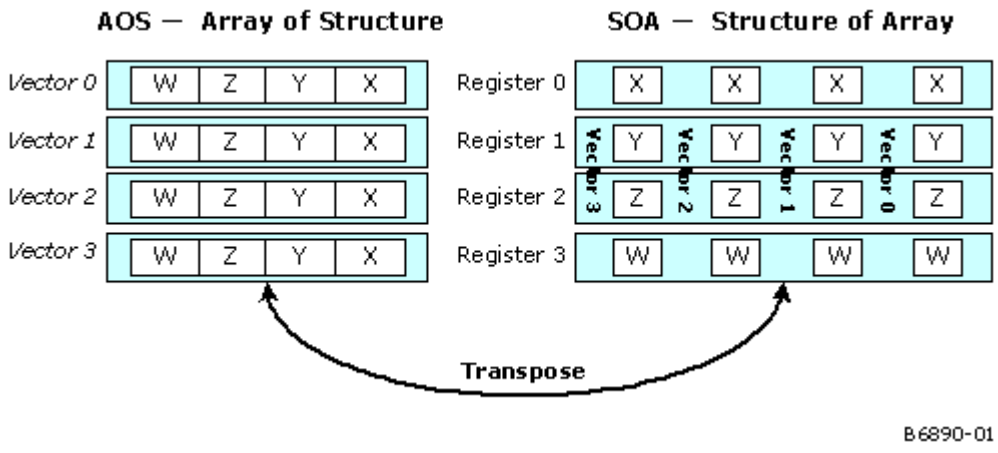
Primary Usage Models

In describing the usage models of the instruction set, the following sections forward reference terminology, syntax, and instructions described later in this specification. For clarity reasons, not all forward references are explained at the point of reference. See the Instruction Set Summary chapter for information about instruction fields and syntax.

AOS and SOA Data Structures

With the Align1 and Align16 access modes, the instruction set provides effective SIMD computation whether data is arranged in array of structures (AOS) form or in structure of arrays (SOA) form. The AOS and SOA data structures are illustrated by the examples in *AOS and SOA Data Structures*. The example shows two different ways of storing four vectors in four SIMD registers. For simplicity, the data vector and the SIMD register both have four data elements. The four data elements in a vector are denoted by X, Y, Z, and W just as for a vertex in 3D geometry. The AOS structure stores one vector in a register and the next vector in another register. The SOA structure stores one data element of each vector in a register and the next element of each vector in the next register and so on. The two structures can be related by a matrix transpose operation.

AOS and SOA Data Structures



3D and media applications take advantage of such broad architecture support and use both AOS and SOA data arrangements.

- Vertices in 3D Geometry (Vertex Shader and Geometry Shader) are arranged in AOS form and use SIMD4x2 and SIMD4 modes, respectively, as detailed below.
- Pixels in 3D Rasterization (Pixel Shader) are arranged in SOA form and use SIMD8 and SIMD16 modes as detailed below.
- Pixels in media are primarily arranged in SOA form, and occasionally in AOS form with possibly mixed modes of operation that uses region-based addressing extensively.

These are preferred methods; alternative arrangements may also be possible. Shared function resources provide data transpose capability to support both modes of operations:

The sampler has a transpose for sample reads, the data port has a transpose for render cache writes, and the URB unit has a transpose for URB writes.

The following 3D graphics API Shader instruction is used in the following sections to illustrate various operation modes:

```
add dst.xyz src0.yxzw src1.zwxy
```

This example is a SIMD instruction that takes two source operands src0 and src1, adds them, and stores the result to the destination operand dst. Each operand contains four floating-point data elements. The data type is determined by the instruction opcode. This instruction also uses source swizzles (.yxzw for src0 and .zwxy for src1) and a destination mask (.xyz). Please refer to the programming specifications of 3D graphics API Shader instructions for more details.

A general register has 256 bits, which can store 8 floating point data elements. For 3D graphics, the mode of operation is (loosely) termed after the data structure as SIMD*m*x*n*, where *m* is the size of the vector and *n* is the number of concurrent program flows executed in SIMD.

Execution with AOS data structures:

- **SIMD4** (short for SIMD4x1) indicates that a SIMD instruction operates on 4-element vectors storedin registers. There is one program flow.
- **SIMD4x2** indicates that a SIMD instruction operates on a pair of 4-element vectors in registers. There are effectively two programs running side by side with one vector per program.

Execution with SOA data structures, also referred to as "channel serial" execution, mostly uses:

- **SIMD8** (short for SIMD1x8) indicates a SIMD instruction based on the SOA data structure where one register contains one data element (the same one) for each of 8 vectors. Effectively, there are 8 concurrent program flows.
- **SIMD16** (short for SIMD1x16) indicates that a SIMD instruction operates on a pair of registers that contain one data element (the same one) for each of 16 vectors. SIMD16 has 16 concurrent program flows.

SIMD16 Mode of Operation

With 16 concurrent program flows, one element of a vector would take two GRF registers. In this mode, two corresponding vectors from the two program flows fill a register.

With the following register mappings,

src0:r2-r9 (with 16 X data elements in r2-r3, Y in r4-5, Z in r6-7 and W in r8-9),

src1:r10-r17,

dst:r18-r25,

the example 3D graphics API Shader instruction can be translated into the following three instructions:

add (16) r18<1>:f r4<8;8,1>:f r14<8;8,1>:f// dst.x = src0.y + src1.z

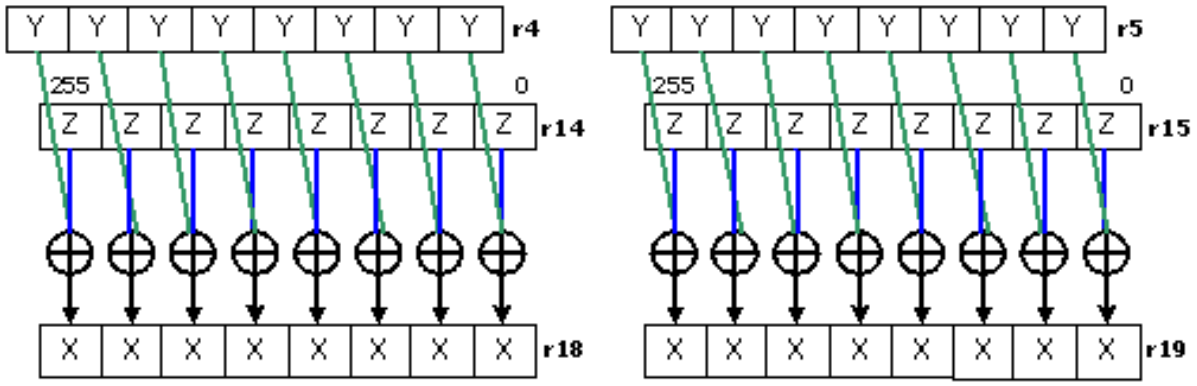
add (16) r20<1>:f r6<8;8,1>:f r16<8;8,1>:f// dst.y = src0.z + src1.w

add (16) r22<1>:f r8<8;8,1>:f r10<8;8,1>:f // dst.z = src0.w + src1.x

The three instructions correspond to the three enabled destination masks as there is no output for the W elements of dst, no instruction is needed for that element. The first instruction inputs the Y elements of src0 and the Z elements of src1 and outputs the X elements of dst. The operation of this instruction is shown in *SIMD16 Mode of Operation*.

With more than one program flow, the above instructions are also subject to the execution mask. The 16-bit dispatch mask is partitioned into four groups with four bits each. For Pixel Shader generated by the Windower, each 4-bit group corresponds to a 2x2 pixel subspan. If a subspan is not valid for a Pixel Shader instance, the corresponding 4-bit group in the dispatch mask is not set. Therefore, the same instructions can be used independent of the number of available subspans without creating bogus data in the subspans that are not valid.

A SIMD16 Example



Similar to SIMD4x2 mode, a constant may also be shared for the 16 program flows. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2 (AOS

format). The example 3D graphics API Shader instruction can then be translated into the following instruction:

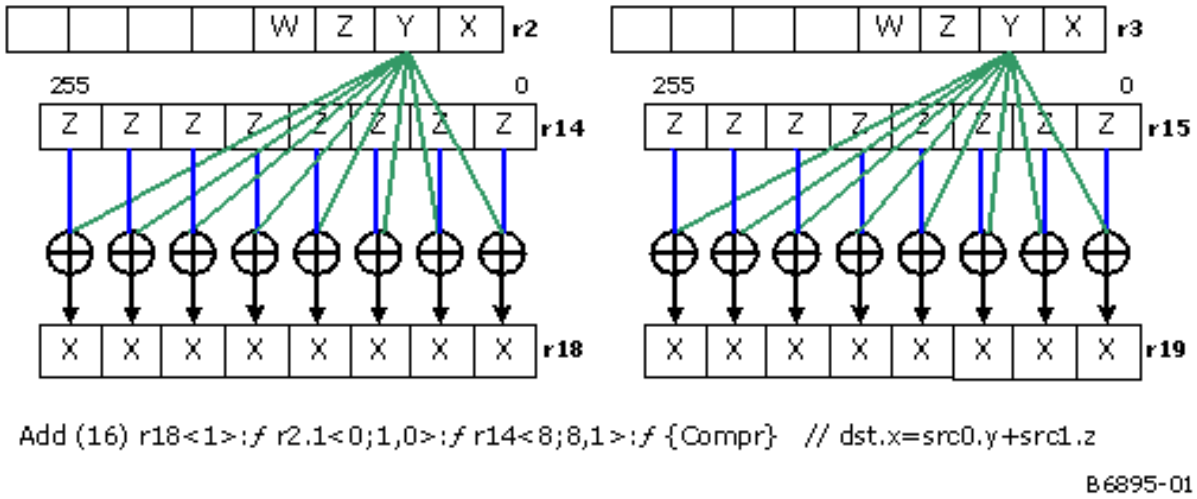
add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr}// dst.x = src0.y + src1.z

add (16) r20<1>:f r2.2<0;1,0>:f r16<8;8,1>:f {Compr}// dst.y = src0.z + src1.w

add (16) r22<1>:f r2.3<0;1,0>:f r10<8;8,1>:f {Compr}// dst.z = src0.w + src1.x

The register region of the first source operand represents a replicated scalar. The operation of the first instruction is illustrated in *SIMD16 Mode of Operation*.

Another SIMD16 Example with an AOS Shared Constant



SIMD8 Mode of Operation

SIMD8 Mode of Operation

Each compressed instruction has two corresponding native instructions. Taking the example instruction shown in Another SIMD16 Example with an AOS Shared Constant, it is equivalent to the following two instructions.

add (8) r18<1>:f r4<8;8,1>:f r14<8;8,1>:f // dst.x[7:0] = src0.y + src1.z

add (8) r19<1>:f r5<8;8,1>:f r15<8;8,1>:f {SecHalf}

// dst.x[15:8] = src0.y + src1.z

Therefore, SIMD8 can be viewed as a special case for SIMD16.

There are other reasons that SIMD8 instructions may be used. Within a program with 16 concurrent program flows, sometime SIMD8 instruction must be used due to architecture restrictions. For example, the address register a0 only have 8 elements, if an indirect GRF addressing is used, SIMD16 instructions are not allowed.

Messages

Communication between the EUs and the shared functions and between the fixed function pipelines (which are not considered part of the "Subsystem") and the EUs is accomplished via packets of information called *messages*. Message transmission is requested via the send instruction. Refer to the send instruction definition in the *ISA Reference* chapter for details.

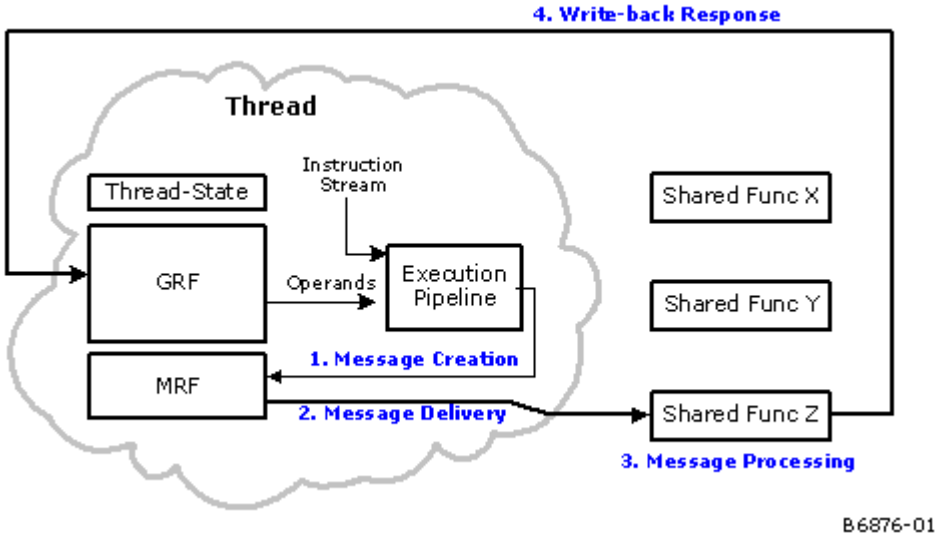
The information transmitted in a message falls into two categories:

- **Message Payload.**
- Associated ("sideband") information provided by:
  - **Message Descriptor.** Specified with the send instruction. Included in the message descriptor is control and routing information such as the target function ID, message payload length, response length, etc.
  - Additional information provided by the send instruction, e.g., the starting destination register number, the execution mask (EMASK), etc.
  - A small subset of Thread State, such as the Thread ID, EUID, etc.

The software view of messages is shown in Data Flow Associated With Messages. There are four basic phases to a message's lifetime as illustrated below:

1. **Creation.**
2. **Delivery.** The thread issues the message for delivery via the send instruction. The send instruction also specifies the destination shared function ID (SFID), and where in the GRF any response is to be directed. The messaging subsystem enqueues the message for delivery and eventually routes the message to the specified shared function.
3. **Processing.** The shared function receives the message and services it accordingly, as defined by the shared function definition.
4. **Writeback.** If called for, the shared function delivers an integral number of registers of data to the thread's GRF in response to the message.

Data Flow Associated With Messages



Message Payload Containing a Header

For most shared functions, the first register of the message payload contains the *header payload* of the message (or simply the *message header*).

Consequently, the rest of the message payload is referred to as the *data payload*.

Messages to Extended Math do not have a header and only contain data payload. Those messages may be referred to as header-less messages. Messages to Gateway combine the header and data payloads in a single message register.

Writebacks

Some messages generate return data as dictated by the 'function-control' (opcode) field of the 'send' instruction (part of the <desc> field). The execution unit and message passing infrastructure do not interpret this field in any way to determine if writeback data is to be expected. Instead, explicit fields in the 'send' instruction to the execution unit the starting GRF register and count of returning data. The execution unit uses this information to set in-flight bits on those registers to prevent execution of any instruction which uses them as an operand until the register(s) is(are) eventually written in response to the message. If a message is not expected to return data, the 'send' instruction's writeback destination specifier (<post_dest>) must be set to 'null' and the response length field of <desc> must be 0 (see 'send' instruction for more details).

The writeback data, if called for, arrives as a series of register writes to the GRF at the location specified by the starting GRF register and length as specified in the 'send' instruction. As each register is written back to the GRF, its in-flight flag is cleared and it becomes available for use as an instruction operand. If a thread was suspended pending return of that register, the dependency is lifted and the thread is allowed to continue execution (assuming no other dependency for that thread remains outstanding).

Message Delivery Ordering Rules

All messages between a thread and an individual shared function are delivered in the ordered they were sent. Messages to different shared functions originating from a single thread may arrive at their respective shared functions out of order.

The writebacks of various messages from the shared functions may return in any order. Further individual destination registers resulting from a single message may return out of order, potentially allowing execution to continue before the entire response has returned (depending on the dependency chain inherent in the thread).

Execution Mask and Messages

The Architecture defines an Execution Mask (EMask) for each instruction issued. This 32b bit-field identifies which SIMD computation channels are enabled for that instruction. Since the 'send' instruction is inherently scalar, the EMask is ignored as far as instruction dispatch is concerned. Further the execution size has no impact on the size of the 'send' instruction's implicit move (it is always 1 register regardless of specified execution size).

The 32b EMask is forwarded with the message to the destination shared function to indicate which SIMD channels were enabled at the time of the 'send'. A shared function may interpret or ignore this field as dictated by the functionality it exposes. For instance, the DataPort writes to the render cache ignoring this field completely, instead it uses the pixel mask included in-band in the message payload to indicate which channels carry valid data.

End-Of-Thread (EOT) Message

The final instruction of all threads must be a *send* instruction that signals 'End-Of-Thread' (EOT). An EOT message is one in which the EOT bit is set in the *send* instruction's 32b <desc> field. When issuing instructions, the EU looks for an EOT message, and when issued, shuts down the thread from further execution and considers the thread completed.

Only a subset of the shared functions can be specified as the target function of an EOT message, as shown in the table below.

Target Shared Functions supporting EOT messages	Target Shared Functions not supporting EOT messages
PixelPort, URB, ThreadSpawner	DataPort

Target Shared Functions supporting EOT messages	Target Shared Functions not supporting EOT messages
PixelPort, URB, ThreadSpawner	DataPort, Sampler

Both the fixed-functions and the thread dispatcher require EOT notification at the completion of each thread. The thread dispatcher and fixed functions in the 3D pipeline obtain EOT notification via the target shared functions.

The thread dispatcher, upon detecting an end-of-thread message, updates its accounting of resource usage by that thread, and is free to issue a new thread to take the place of the ended thread. Fixed functions require end-of-thread notification to maintain accounting as to which threads it issued have completed and which remain outstanding, and their associated resources such as URB handles.

Unlike the thread dispatcher, fixed-functions discriminate end-of-thread messages, only acting upon those from threads which they originated, as indicated by the 4b fixed-function ID present in R0 of end-of-thread message payload. This 4b field is attached to the thread at new-thread dispatch time and is placed in its designated field in the R0 contents delivered to the GRF. Thus, to satisfy the inclusion of the fixed-function ID, the typical end-of-thread message generally supplies R0 from the GRF as the first register of an end-of-thread message.

As an optimization, an end-of-thread message may be overload upon another "productive" message, saving the cost in execution and bandwidth of a dedicated end-of-thread message. Outside of the end-of-thread message, most threads issue a message just prior to their termination (for instance, a Pixel Port write to the framebuffer) so the overloaded end-of-thread is the common case. The requirement is that the message contains R0 from the GRF (to supply the fixed-function ID), and that destination shared function be either (a) the URB; (b) the Pixel Port; or, (c) the Thread Spawner, as these functions reside on the O-Bus. In the case where the last real message of a thread is to some other shared function, the thread must issue a separate message for the purposes of signaling end-of-thread.

Performance

The Architecture imposes no requirement as to a shared function's latency or throughput. Due to this as well as factors such as message queuing, shared bus arbitration, implementation choices in bus bandwidth, and instantaneous demand for that function, the latency in delivering and obtaining a response to a message is non-deterministic. It is expected that an implementation has some notion of fairness in transmission and servicing of messages so as to keep latency outliers to a minimum.

Other factors to consider with regard to performance:

Software prefetching techniques may be beneficial for long latency data fetches (i.e., issue a load early in the thread for data that is required late in the thread).

Message Description Syntax

All message formats are defined in terms of DWords (32 bits). The message in all cases has the same width as general-purpose register. The registers and DWords within the message are named as follows, where n is the register number, and d is the DWord number from 0. For writeback messages, the register number indicates the offset from the specified starting destination register.

Dispatch Messages: **R**n.d

Dispatch messages are sent by the fixed functions to dispatch threads. See the fixed function chapters in the *3D and Media* volume.

SEND Instruction Messages: **M**n.d

These are the messages initiated by the thread via the SEND instruction to access shared functions. See the chapters on the shared functions later in this volume.

Writeback Messages: **W**n.d

These messages return data from the shared function to the GRF where it can be accessed by thread that initiated the message.

The bits within each DWord are given in the second column in each table.

Message Errors

Messages are constructed via software, and not all possible bit encodings are legal, thus there is the possibility that a message may be sent containing one or more errors in its descriptor or payload contents. There are two points of error detection in the message passing system: (a) the message delivery subsystem is capable of detecting bad FunctionIDs and some cases of bad message lengths; (b) the shared functions contain various error detection mechanisms which identify bad sub-function codes, bad message lengths, and other misc. errors. The error detection capabilities are specific to each shared function. The execution unit hardware itself does not perform message validation prior to transmission.

In both cases, information regarding the erroneous message is captured and made visible through MMIO registers, and the driver notified via an interrupt mechanism.

The set of possible errors is listed in Error Cases with the associated outcome.

Error Cases

Error	Outcome
Bad Shared Function ID	The message is discarded before reaching any shared function. If the message specified a destination, those registers will be marked as in-flight, and any future usage by the thread of those registers will cause a dependency which will never clear, resulting in a hung thread and eventual time-out.
Unknown opcode Incorrect message length	The destination shared function detects unknown opcodes (as specified in the 'send' instructions <desc> field) and known opcodes where the message payload is either too long or too short and threats these cases as errors. When detected, the shared function latches and makes available via MMIO registers the following information: the EU and thread ID which sent the message, the length of the message and expected response, and any relevant portions of the first register (R0) of the message payload. The shared function alerts the driver of an erroneous message through, and interrupt mechanism then continues normal operation with the subsequent message.
Bad message contents in payload	Detection of bad data is an implementation decision of the shared function. Not all fields may be checked by the shared function, so an erroneous payload may return bogus data or no data at all. If an erroneous value is detected by the shared function, it is free to discard the message and continue with the subsequent message. If the thread was expecting a response, the destination registers specified in the associated 'send' instruction are never cleared potentially resulting in a hung thread and time-out.

Error	Outcome
Incorrect response length	Case: too few registers specified - the thread may proceed with execution prior to all the data returning from the shared function, resulting in the thread operating on bad data in the GRF. Case: too many registers specified - the message response does not clear all the registers of the destination. In this case, if the thread references any of the residual registers, it may hand and result in an eventual time-out.
Improper use of End-Of-Thread (EOT)	Any 'send' instruction which specifies EOT must have a 'null' destination register. The EU enforces this and, if detected, will not issue the 'send' instruction, resulting in a hung thread and an eventual time-out. The 'send' instruction specifies that EOT is only recognized if the <desc> field of the instruction is immediate. Should a thread attempt to end a thread using a <desc> sourced from a register, the EOT bit will not be recognized. In this case, the thread will continue to execute beyond the intended end of thread, resulting in a wide range of error conditions.
Two outstanding messages using overlapping GRF destinations ranges	This is not checked by HW. Due to varying latencies between two messages, and out-of-order, non-contiguous writeback cycles, the outcome in the GRF is indeterminate; may be the result from the first message, or the result from the second message, or a combination of both.

Registers and Register Regions

Register Files

Registers are grouped into different name spaces called register files. There are two register files, the General Register File and the Architecture Register File. A third encoding of some register file instruction fields indicates immediate operands within instructions rather than a register file.

- General Register File (GRF): The GRF contains general-purpose read-write registers.
- Architecture Register File (ARF): The ARF contains all architectural registers defined for specific purposes, including address registers (*a#*), accumulators (*acc#*), flags (*f#*), notification count (*n#*), instruction pointer (*ip*), null register (*null*), etc.
- Immediate: Certain instructions can take immediate source operands. A distinct register file field encoding indicates an immediate operand.

Each thread executed in an EU has its own thread context, a dedicated register space that is not shared between threads, whether executing on a common EU or on a different EU. In the rest of the chapters in this volume, register access is relative to a given thread.

GRF Registers

Number of Registers:	Various
Default Value:	None
Normal Access:	RW
Elements:	Various
Element Size:	Various
Element Type:	Various
Access Granularity:	Byte
Write Mask Granularity:	Byte
Indexable?	Yes

Registers in the General Register File are the most commonly used read-write registers. During the execution of a thread, GRF registers are used to store the temporary data, and serve as the destination to receive data from shared function units (and sometimes from a fixed function unit). They are also used to store the input (initialization) data when a thread is created. By allowing fixed function hardware to initialize some portion of GRF registers during thread dispatch time, architecture can achieve better parallelism. A thread's execution efficiency can also be improved as some data are already in the register to be executed upon. Besides these registers containing thread's payload, the rest of GRF registers of a thread are not initialized.

Summary of GRF Registers

Register File	Register Name	Description
General Register File (GRF)	r#	General purpose read write registers

Each execution unit has a fixed size physical GRF register RAM. The GRF register RAM is shared by all threads on the EU. Each thread has a dedicated space of 128 or 256 register, r0 through r127 or r255.

GRF registers can be accessed using region-based addressing at byte granularity (both read and write). A source operand must be contained within two adjacent registers. A destination operand must be contained within one register. GRF registers support direct addressing and register-indirect addressing. Register-indirect addressing uses the address registers (ARF registers a#) and an immediate address offset value.

When accessing (read and/or write) outside the GRF register range allocated for a given thread either through direct or indirect addressing, the result is unpredictable.

Register Size	Allocation Granularity	Number per Thread
256 bits	Static allocation of 128 and 256 registers	128/256 registers

ARF Registers

ARF Registers Overview

GRF registers are directly indicated by a unique register file encoding. Besides registers that are directly indicated by a unique register file coding, all other registers belong to the Architecture Register File (ARF). Encodings of architecture register types are based on the MSBs of the register number field, RegNum, in the instruction word. The RegNum field has 8 bits.

The 4 MSBs, RegNum[7:4], represent the Architecture **Register Type**. This is summarized in the *Summary of Architecture Registers* table below.

RegNum[3:0], represent the Architecture **Register Number**.

Summary of Architecture Registers

Register Type	Register Name	Description
0000b	*null*	Null register
0001b	*a0.#*	Address register
0010b	*acc#*	Accumulator register
0011b	*f#.#*	Flag register
0100b	*ce#*	Channel Enable register
0111b	*sr0.#*	State register
1000b	*cr0.#*	Control register
1001b	*n#*	Notification Count register
1010b	*ip*	Instruction Pointer register
1011b	*tdr*	Thread Dependency register
1100b	*tm0*	Pause register
1101b	*fc#.#*	Flow Control register
1110b	*Reserved*	Reserved

The remaining **Register Number** field RegNum[3:0] is used to identify the register number of a given architecture register type. Therefore, the maximum number of registers for a given architecture register type is limited to 16. The subregister number field, SubRegNum, in the instruction word has 5 bits. It is used to address subregister regions for an architecture register supporting register subdivision.

The SubRegNum field is in units of bytes. Therefore, the maximum number of bytes of an architecture register is limited to 32. Depending on the alignment restriction of a register type, only certain encodings of SubRegNum field apply for an architecture register. The detailed definitions are provided in subsequent sections.

In general, an ARF register can be dst (destination) or src0 (source 0, first source operand) for an instruction. Depending on the register and the instruction, other restrictions may apply.

Access Granularity

ARF registers may be accessed with subregister granularity according to the descriptions below and following the same rule of region-based addressing for GRF.

The machine code for register number and subregister number of ARF follows the same rule as for other register files with byte granularity. For an ARF as a source operand, the region-based address controls the source swizzle mux. The destination subregister number and destination horizontal stride can be used to generate the destination write mask at byte level.

Subregister fields of an ARF register may not all be populated (indicated by the subregister being indicated as reserved). Writes to unpopulated subregisters are dropped; there are no side effect. Reads from unpopulated subregisters, if not specified, return unpredictable data.

Some ARF registers are read-only. Writes to read-only ARF registers are dropped and there are no side effects.

Null Register

Null Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0000b
Number of Registers:	1
Default Value:	N/A
Normal Access:	N/A
Elements:	N/A
Element Size:	N/A
Element Type:	N/A
Access Granularity:	N/A
Write Mask Granularity:	N/A
SecHalf Control?	N/A
Indexable?	No

The null register is a special encoding for an operand that does not have a physical mapping. It is primarily used in instructions to indicate non-existent operands. Writing to the null register has no side effect. Reading from the null register returns an undefined result.

The null register can be used where a source operand is absent. For example, for a single source operand instruction such as MOV or NOT, the second source operand src1 must be a null register.

When the null register is used as the destination operand of an instruction, it indicates the computed results are not stored in any registers. However, implied writes to the accumulator register, if applicable, may still occur for the instruction. When the conditional modifier is present, updates to the selected flag register also occur. In this case, the register region fields of the 'null' operand are valid.

Another example use is to use the null register as the posted destination of a *send* instruction for data write to indicate that no write completion acknowledgement is required. In this case, however, the register region fields are still valid. The null register can also be the first source operand for a send instruction indicating the absent of the implied move. See the *send* instruction for details.

Address Register

Address Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0001b
Number of Registers:	1
Default Value:	None
Normal Access:	RW
Elements:	16
Element Size:	16 bits
Element Type:	UW or UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	N/A
Indexable?	No

There are sixteen address subregisters forming a 16-element vector. Each address subregister contains 16 bits. Address subregisters can be used as regular source and destination operands, as the indexing addresses for register-indirect-addressed access of GRF registers, and also as the source of the message descriptor for the *send* instruction.

Register and Subregister Numbers for Address Register

RegNum[3:0]	SubRegNum[4:0]
0000b = a0 All other encodings are reserved.	When register a0 or subregisters in a0 are used as the address register for register-indirect addressing, the address subregisters must be accessed as unsigned word integers. Therefore, the subregister number field must also be word-aligned. 00000b = **a0.0:uw** 00010b = **a0.1:uw** 00100b = **a0.2:uw** 00110b = **a0.3:uw** 01000b = **a0.4:uw** 01010b = **a0.5:uw** 01100b = **a0.6:uw** 01110b = **a0.7:uw**

RegNum[3:0]	SubRegNum[4:0]
	10000b = **a0.8:uw** 10010b = **a0.9:uw** 10100b = **a0.10:uw** 10110b = **a0.11:uw** 11000b = **a0.12:uw** 11010b = **a0.13:uw** 11100b = **a0.14:uw** 11110b = **a0.15:uw** All other encodings are reserved. However, when register a0 or subregisters in a0 are explicit source and/or destination registers, other data types are allowed as long as the register region falls in the 128-bit range.

Address Register Fields

DWord	Bits	Description
7	31:16	**Address subregister a0.15:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.14:uw**. Follows the same format as **a0.2**.
6	31:16	**Address subregister a0.13:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.12:uw**. Follows the same format as **a0.2**.
5	31:16	**Address subregister a0.11:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.10:uw**. Follows the same format as **a0.2**.
4	31:16	**Address subregister a0.9:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.8:uw**. Follows the same format as **a0.2**.
3	31:16	**Address subregister a0.7:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.6:uw**. Follows the same format as **a0.2**.
2	31:16	**Address subregister a0.5:uw**. Follows the same format as **a0.3**.
	15:0	**Address subregister a0.4:uw**. Follows the same format as **a0.2**.
1	31:16	**Address subregister a0.3:uw**. This field can be used for register-indirect register addressing or serve as extended descriptor for a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *send* instruction, it provides the higher 16 bits of <exdesc>. Format: U12 or U16
	15:0	**Address subregister a0.2:uw**. This field can be used for register-indirect register addressing or serve as extended descriptor for a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *send* instruction, it provides the lower 16 bits of

DWord	Bits	Description
		<exdesc>. Format: U12 or U16
0	31:16	**Address subregister a0.1:uw**. This field can be used for register-indirect register addressing or serve as message descriptor or extended descriptor for a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *send* instruction, it provides the higher 16 bits of <desc> or <exdesc>. Format: U12 or U16.
	15:0	**Address subregister a0.0:uw**. This field can be used for register-indirect register addressing or serve as message descriptor or extended descriptor for a *send* instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a *send* instruction, it provides the lower 16 bits of <desc> or <exdesc>. Format: U12 or U16.

When used as a source or destination operand, the address subregisters can be accessed individually or as a group. In the following example, the first instruction moves 8 address subregisters to the first half of GRF register r1, the second instruction replicates a0.4:uw as an unsigned word to the second half of r1, the third instruction moves the first 4 words in r1 into the first 4 address subregisters, and the fourth instruction replicates r1.4 as a unsigned word to the next 4 address subregisters.

```
mov (8) r1.0<1>:uw a0.0<8;8,1>:uw // r1.n = a0.n for n = 0 to 7 in words
mov (8) r1.8<1>:uw a0.4<0;1,0>:uw // r1.m = a0.4 for m = 8 to 15 in words
mov (4) a0.0<1>:uw r1.0<4;4,1>:uw // a0.n = r1.n for n = 0 to 3 in words
mov (4) a0.4<1>:uw r1.4<0;1,0>:uw // a0.m = r1.4 for m = 4 to 7 in words
```

When used as the register-indirect addressing for GRF registers, the address subregisters can be accessed individually or as a group. When accessed as a group, the address subregisters must be group-aligned. For example, when two address subregisters are used for register indirect addressing, they must be aligned to even address subregisters. In the following example, the first instruction is legal. However, the second one is not. As ExecSize = 8 and the width of src0 is 4, two address subregisters are used as row indices, each pointing to 4 data elements spaced by HorzStride = 1 dword. For the first instruction, the two address subregisters are a0.2:uw and a0.3:uw. The two align to a DWord group in the address register. However, the two address subregisters for the second instruction are a0.3:uw and a0.4:uw. They are not DWord-aligned in the address register and therefore violate the above-mentioned alignment rule.

```
mov (8) r1.0<1>:d r[a0.2]<4,1>:d // a0.2 and a0.3 are used for src1
mov (8) r1.0<1>:d r[a0.3]<4,1>:d // ILLEGAL use of register indirect
```

Programming Note	
Context:	ARF Registers
Implementation restriction: When used as the source operand <desc> for the send instruction, only the first dword subregister of a0 register is allowed (i.e. a0.0:ud, which can be viewed as the combination of a0.0:uw and a0.1:uw). In addition, it must be of UD type and in the following form <desc> = a0.0<0;1,0>:ud.	

Programming Note
When trying to decode index registers for uninitialized channels for VXH indirect instructions, hangs in Silicon are detected. Address registers must be initialized to 0 in kernels before using them for indirect addressing.

Programming Note	
Context:	ARF Registers
Performance Note: There is only one scoreboard for the whole address register. When a write to some subregisters is in flight, hardware stalls any instruction writing to other subregisters. Software may use the destination dependency control {NoDDChk, NoDDClr} to improve performance in this case. Similarly, when a write to some subregisters is in flight, hardware stalls any instruction sourcing other subregisters until the write retires.	

Description
Implementation Restriction on Register Access: When the address registers are used as source and/or destination, hardware does not ensure prevention of write after read hazard across execution pipes. Software must ensure that the address registers are read before it is overwritten by an instruction by inserting dependencies.

Accumulator Registers

Accumulator Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	4+8
Default Value:	None
Normal Access:	RW
Elements:	8
Element Size:	33 bits
Element Type:	UW & UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	No
Indexable?	No

Accumulator registers can be accessed either as explicit or implied source and/or destination registers. To a programmer, each accumulator register may contain either 8 DWords or 16 Words of data elements. However, as described in the Implementation Precision Restriction notes below, each data element may have higher precision with added guard bits not indicated by the numeric data type.

Accumulator capabilities vary by data type, not just data size, as described in the Accumulator Channel Precision table below. For example, D and F are both 32-bit data types but differ in accumulator support.

See the Accumulator Restrictions section for information about additional general accumulator restrictions and also accumulator restrictions for specific instructions.

Register and Subregister Numbers for Accumulator Registers

RegNum[3:0]	SubRegNum[4]	SubRegNum[3:0]
0000b-0011b = acc0-acc3 1000b-1111b = mme0-mme7 All other encodings are reserved.	for acc#: 0 : Lower half 1 : Upper half for mme#: Reserved	Reserved: MBZ

-

Description
Accumulators are updated implicitly only if the AccWrCtrl bit is set in the instruction. The Accumulator Disable bit in control register cr0.0 allows software to disable the use of AccWrCtrl for implicit accumulator updates. The write enable in word granularity for the instruction is used to update the accumulator. Data in disabled channels is not updated.
When an accumulator register is an implicit source or destination operand, hardware always uses acc0 by default and also uses acc1 if the execution size exceeds the number of elements in acc0. When implicit access to acc1 is required, QtrCtrl is used. Note that QtrCtrl can be used only if acc1 is accessible for a given data type. If acc1 is not accessible for a given data type, QtrCtrl defaults to acc0.

acc0 and acc1 are supported for half-precision (HF, Half Float) and single precision (F, Float). Use QtrCtrl of Q2 or Q4 to access acc1 for Float. use QtrCtrl of H2 to access acc1 for Half Float.
Examples:
`// Updates acc0 and acc1 because it is SIMD16:`
`add (16) r10:f r11:f r12:f {AccWrEn}`
`// Updates acc0 because it is SIMD8:`
`add (8) r10:f r11:f r12:f {AccWrEn}`
`// Updates acc1. Implicit access to acc1 using QtrCtrl:`
`add (8) r10:f r11:f r12:f {AccWrEn, Q2}`
`// Updates acc1 for Half Floats using QtrCtrl:`
`add (16) r10:hf r11:hf r12:hf {AccWrEn, H2}`

<table>
<tr><th>Description</th></tr>
<tr><td>Accumulator registers are defined in pairs (i.e., acc0-1 and acc2-3) and read/write access cannot span across these pairs.</td></tr>
<tr><td>Accumulator registers may be accessed explicitly on src0 and src1 operand.</td></tr>
<tr><td>When accumulator is used as src1 operand the following applies
1. Source modifier must not be used with integer datatypes.
2. Precision of accumulator is restricted to same as GRF, implying that sign bit will come from the MSB of the datatype bits.</td></tr>
<tr><td>When destination is accumulator with offset 0, destination horizontal stride must be 1.</td></tr>
<tr><td>The following datatypes conversions are supported with Accumulator on source or destination.
<table>
<tr><th>Dst</th><th>Src0/Src1</th></tr>
<tr><td>*W/*D</td><td>*W/*D</td></tr>
<tr><td>HF/F</td><td>HF/F</td></tr>
<tr><td>DF</td><td>DF</td></tr>
</table>
</td></tr>
</table>

- Register Regioning patterns where register data bit locations are changed between source and destination are not supported when an accumulator is used as an implicit source or an explicit source in an instruction.
- Reading accumulator content with a datatype different from the previous write will result in undeterministic values.
- Word datatype write to accumulator is not allowed when destination is odd offset strided by 2.
- For any DWord operation, including DWord multiply, accumulator can store up to 8 channels of data, with only acc0 supported.
- When an accumulator register is an explicit destination, it follows the rules of a destination register. If an accumulator is an explicit source operand, its register region must match that of the destination register with the exception(s) described below.

Implementation Precision Restriction: As there are only 64 bits per channel in DWord mode (D and UD), it is sufficient to store the multiplication result of two DWord operands as long as the post source modified sources are still within 32 bits. If any one source is type UD and is negated, the negated result becomes 33 bits. The DWord multiplication result is then 65 bits, bigger than the storage capacity of accumulators. Consequently, the results are unpredictable.

Implementation Precision Restriction: As there are only 33 bits per channel in Word mode (W and UW), it is sufficient to store the multiplication result of two Word operands with and without source modifier as the result is up to 33 bits. Integers are stored in accumulator in 2's complement form with bit 32 as the sign bit. As there is no guard bit left, the accumulator can only be sourced once before running into a risk of overflowing. When overflow occurs, only modular addition can generate a correct result. But

in this case, conditional flags may be incorrect. When saturation is used, the output is unpredictable. This is also true for other operations that may result in more than 33 bits of data. For example, adding UD (FFFFFFFFh) with D (FFFFFFFFh) results in 1FFFFFFFEh. The sign bit is now at bit 34 and is lost when stored in the accumulator. When it is read out later from the accumulator, it becomes a negative number as bit 32 now becomes the sign bit.

Accumulator Channel Precision

Data Type	Accumulator Number	Number of Channels	Bits Per Channel	Description
DF	acc0-acc3	4	64	When accumulator is used for Double Float, it has the exact same precision as any GRF register. When accessing multiple accumulators in a single instruction even accumulator number must be used.
F	acc0-acc3	8	32	When accumulator is used for Float, it has the exact same precision as any GRF register. When accessing multiple accumulators in a single instruction even accumulator number must be used.
HF	acc0-acc3	16	16	When accumulator is used for Half Float, it has the exact same precision as any GRF register. When accessing multiple accumulators in a single instruction even accumulator number must be used.
BF	N/A	N/A	N/A	Not supported data type
Q	N/A	N/A	N/A	Not supported data type.
D (UD)	acc0/acc2	8	33/64	When the internal execution data type is doubleword integer, each accumulator register contains 8 channels of (extended) doubleword integer values. The data are always stored in accumulator in 2's complement form with 64 bits total regardless of the source data type. This is sufficient to construct the 64-bit D or UD multiplication results using an instruction macro sequence consisting of *mul*, *mach*, and *shr* (or *mov*). Writing to acc1/acc3 using any datatype may corrupt this result.
W (UW)	acc0/acc2	16	33	When the internal execution data type is word integer, each accumulator register contains 16 channels of (extended) word integer values. The data are always stored in accumulator in 2's complement form with 33 bits total. This supports single instruction multiplication of two-word sources in W and/or UW format. Writing to acc1/acc3 using any datatype may corrupt this result.

Data Type	Accumulator Number	Number of Channels	Bits Per Channel	Description
B (UB)	N/A	N/A	N/A	Not supported data type.

Description
Implementation Restriction on Register Access: When the accumulator registers are used as source and/or destination, hardware does not ensure prevention of write after read hazard across execution pipes. Software must ensure that the accumulator registers are read before it is overwritten by an instruction by inserting dependencies.

Math Macro Extended Accumulators
Math Macro Extended mme0-mme7

These are accumulator registers defined for a special purpose. They are used to emulate IEEE-compliant fdiv and sqrt macro-operations. The access is different from *acc0* and *acc1*, which are defined as full 256-bit registers having 8 DWords and may be accessed explicitly or implicitly. Conversely, these math macro extended accumulators consist of just a few bits and have very restricted access.

- These registers are accessed directly by math macro opcodes only. **Note:** These macro operations are *madm* and some others defined under the *math* opcode section. The macro descriptions also define the restrictive implicit uses of these registers.
- Math macro sequences (e.g., *madm* and certain *math* ops) store and load additional bits of precision into special per-thread registers (mme#). The low bits of the SubRegNum field in the instruction bits are repurposed to represent which. The *nomme* value is specified when no write to a math macro extended register is required; think of it as a writing to null. An example operand syntax might be *r12.mme4:f* meaning the access uses both *r12* and *mm4* to represent to the fully extended precision value.

MathMacroExt[3:0]	Register
0000b	mme0
0001b	mme1
0010b	mme2
0011b	mme3
0100b	mme4
0101b	mme5
0110b	mme6
0111b	mme7
1000b	nomme

- Note: *MathMacroExt* was referred to previous generations as *SpecialAcc*. And the syntax (*acc2* to *acc9* and *noacc* were used prior).

Flag Register

Flag Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0011b
Number of Registers:	2
Default Value:	None
Normal Access:	RW
Elements:	2
Element Size:	16 bits
Element Type:	UD & UW
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	Yes
Indexable?	No

Each flag register contains two 16-bit subregisters. Each flag bit corresponds to a data channel. Predication uses flag values to enable or disable channels. Conditional modifiers assign flag values. If an instruction uses both predication and conditional modifiers, both features use the same flag register or subregisters.

Flags can be split to halves, quarters, or eighths using the QtrCtrl and NibCtrl instruction fields. Those fields affect the selection of flags for predication and conditional modifiers, but do not affect reading or writing flags as explicit instruction operands.

The values held in the individual bits of a flag register are the result of the most recent instruction with a conditional modifier and specifying that flag register. For example:

```
add.nz.f0.0 ...
```

Updates flag subregister f0.0 with the per-channel results of the not zero condition.

The flag register has per-bit write enables. When being updated as the secondary destination associated with a conditional modifier, only the bits corresponding to the enabled channels in *EMask* are updated. Other bits in the flag subregister are unchanged.

Flag registers and subregisters can also be explicit source or destination operands.

The *sel* instruction does not update flags.

Register and Subregister Numbers for Flag Register

RegNum[3:0]	SubRegNum[4:0]
0000b = **f0:ud** 0001b = **f1:ud** Other encodings are reserved.	00000b = **f*n*.0:uw** 00010b = **f*n*.1:uw** Other encodings are reserved.

f0:ud or f1:ud etc reference an entire flag register. f0.0:uw, f0.1:uw, fn.0:uw, and fn.1:uw reference the flag subregisters.

Channel Enable Register

Channel Enable Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0100b
Number of Registers:	1
Default Value:	N/A
Normal Access:	RO
Elements:	1
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	N/A
SecHalf Control?	No
Indexable?	No

Register and Subregister Numbers for Channel Enable Register

RegNum[3:0]	SubRegNum[4:0]
0000b = ce All other encodings are reserved.	00000b = **ce:ud**. All other encodings are reserved.

Channel Enable Register Fields

DWord	Bits	Description
0 (ce:ud)	31:0	**Channel Enable Register** Format: U32 This field contains 32 bits of Channel Enables for the current instruction.

State Register

State Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0111b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	4
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Byte
Write Mask Granularity:	N/A
SecHalf Control?	No
Indexable?	No

Register and Subregister Numbers for State Register

RegNum[3:0]	SubRegNum[4:0]
0000b = sr0 All other encodings are reserved.	Valid encoding range: 00000b = **sr0.0:ud** 00100b = **sr0.1:ud** 01000b = **sr0.2:ud** 01100b = **sr0.3:ud** All other encodings are reserved.

State Register Fields

DWord	Bits	Description
0 (sr0.0:ud)	31:28	**Reserved**. MBZ.
	27:24	**FFID** (Fixed Function Identifier). Specifies which fixed function unit generates the current thread. This field is set at thread dispatch and is forwarded on the message bus for all out-going messages from this thread. Initialized at dispatch. Read-only.
	23	**Priority Class**. This field, when set, indicates the thread belongs to the high priority class, which has higher scheduling priority over any thread with this field cleared. The priority field below determines the relative priority within the same priority class. This field is initialized by the thread dispatcher at thread dispatch time and stays unchanged throughout the life span of the thread.

DWord	Bits	Description
		This field is forwarded on the message bus to the message bus arbiter for all out-going messages. It serves as a priority hint for the target shared function. See the Shared Function chapters for whether and how a shared function uses this priority hint. This bit is honored only when Threads Execution Arbitration Mode is set to Age Based. 0 = Low priority class. 1 = High priority class. Initialized at dispatch. Read-only.
	22:20	**Priority**. This field is the relative aging priority of the thread. This field indicates the 'age' of this thread relative to other threads within the EU. No two threads in the same EU can have the same priority number (independent of the priority class value). Within the same priority class, an older thread (with a larger priority number) has higher schedule priority over a younger thread. This field is set to zero at a thread's dispatch. During a thread's run time, this field may or may not be incremented when a new thread is dispatched to the same EU. It is only incremented when another thread's priority number is incremented and reaches the same value. For example, if currently there is a thread with priority 0 on an EU, then dispatching a new thread to that EU causes the old thread's priority number to increment to 1. However, if the active thread (assuming for simplicity that there is only one) on an EU has a priority number 1 (or 2 or 3), then dispatching a new thread to this EU does not change the old thread's priority number. As threads on an EU may terminate in arbitrary order, the exact number for a thread depends on the dynamic execution of threads. When thread context is saved and restored after pre-emption, the Priority is not restored to the original state. Instead, the priority is initiated as if new threads were loaded. Initialized to 0. Read-only.
	19:13	**Reserved**. MBZ.
	19:14	**Reserved**. MBZ.
	13:4	[13:11] Slice ID. [10:9] Dual-SubSlice ID [8] SubSlice ID. [7] : EUID[2] [6] : Reserved [5:4] EUID[1:0] Initialized by hardware. Read-only.
	3	**Reserved**. MBZ.
	2:0	**TID** (The thread identifier). Specifies the thread slot that the current thread is assigned to. This field is set at thread dispatch.

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td></td><td>Initialized by hardware. Read-only.</td></tr>
<tr><td rowspan="4">1
(sr0.1:ud)</td><td>31:21</td><td>FFTID (Fixed Function Thread ID). There is no connection between this thread ID, assigned in fixed functions, and the TID assigned in the EUs.</td></tr>
<tr><td>31:21</td><td>FFTID (Fixed Function Thread ID). There is no connection between this thread ID, assigned in fixed functions, and the TID assigned in the EUs.
Initialized at dispatch from R0. Read-only.</td></tr>
<tr><td>15:7</td><td>Reserved. MBZ</td></tr>
<tr><td>5:0</td><td>IEEE Exception. The exception bits are sticky bits set by the opcode when one of the exception is triggered. These bits are defined per thread and all channels update one sticky bit. These bits may be cleared by software or on a thread load. Updates to these bits may be turned OFF by the IEEE Exception trap enable in the CR register. When these bits are required as source of an operation, the previous instruction must use a {Switch}. This ensures all asynchronous flag updates are complete before using as source operand. The following table describes these bits:
Workaround: These bits will have undefined value if a previously saved GPGPU context is restored for execution. All new contexts will have these bits initialized to zero.
<table>
<tr><th>Bits</th><th>Definition</th></tr>
<tr><td>5</td><td>Reserved</td></tr>
<tr><td>4</td><td>Inexact Exception</td></tr>
<tr><td>3</td><td>Overflow</td></tr>
<tr><td>2</td><td>Underflow</td></tr>
<tr><td>1</td><td>Divide by Zero</td></tr>
<tr><td>0</td><td>Invalid Operation</td></tr>
</table>
<table>
<tr><th>Programming Note</th></tr>
<tr><td>Errata: Programs with mov, csel, cmpn or non-raw sel opcodes may not have correct IEEE invalid flag set when source is SNAN.</td></tr>
</table>
Initialized to 0. Read/Write</td></tr>
<tr><td>2
(sr0.2:ud)</td><td>31:0</td><td>Dispatch Mask (DMask). This 32-bit field specifies which channels are active at Dispatch time. This field is used by hardware to initialize the mask register.
Format: U32
Initialized at dispatch. Read/Write.</td></tr>
</table>

DWord	Bits	Description
3 (sr0.3:ud)	31:0	**Vector Mask** (VMask). This 32-bit field contains, for each 4-bit group, the OR of the corresponding 4-bit group in the dispatch mask. This field is used by hardware to initialize the mask register. Format: U32 Initialized at dispatch. Read-only.

Implementation Restriction on Register Access: When the state register is used as a source and/or destination, hardware does not ensure execution pipeline coherency. Software must make the immediate next instruction dependent on an instruction that uses state register as an explicit operand. This is important as the state register is an implicit source or destination for many instructions. For example, fields like IEEE Exception may be an implicit destination updated by multiple back-to-back instructions. Therefore, if the instructions updating the state register doesn't make the next instruction stall, subsequent instructions may have undefined results.

Control Register

Control Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1000b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW/PW
Elements:	4
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control?	No
Indexable?	No

The Control register is a read-write register. It contains four 32-bit subregisters that can be accessed individually.

Subregister *cr0.0:ud* contains normal operation control fields such as the floating-point mode and the accumulator disable. It also contains the primary exception status/control field that allows software to switch back to the application thread from the System Routine.

Subregister *cr0.1:ud* contains the mask and status/control fields for all exceptions. The exception fields are arranged in significance-decreasing order from MSB to LSB. This arrangement allows the System Routine to use the *lzd* instruction to find the high priority exceptions and handle them first. As each exception is mapped to a single bit, another exception priority order may be implemented by software. The System Routine may choose to handle one exception at a time, by handling the exception detected

by an *lzd*instruction and returning to the application thread. Or it may choose to handle all the concurrent exceptions, by looping through the exception fields until all outstanding exceptions are handled before returning back to the application thread.Exception enable bits (bits 15:0 in *cr0.1:ud*) control whether an exception causes hardware to jump to the System Routine or not. Exception status and control bits (bits 31:16 in *cr0.1:ud*) indicate which exceptions have occurred, and are used by the system routine to clear the exception. Even if a given exception is disabled, the corresponding exception status and control bit still reflects its status, whether an exception event has occurred or not.

cr0.2:ud contains the **Application IP (AIP)** indicating the current thread IP when an exception occurs.

cr0.3:ud is reserved. Values written to this subregister are dropped; the result of reading from this subregister is unpredictable.

Fields in Control registers also reference a virtual register called **System IP (SIP)**. SIP is the virtual register holding the global System IP, which is the initial instruction pointer for the System Routine. The SIP is a GraphicsAddress. There is only one SIP for the whole system. It is virtual only from a thread's point of view, as it is not visible (i.e. not readable and not writeable) to the thread software executed on an EU. It can only be accessed indirectly by the hardware to respond to exception events. Upon an exception, hardware performs some bookkeeping (e.g. saving the current IP into AIP) and then jumps to SIP. Upon finishing exception handling, the System Routine may return back to the application by clearing the Primary Exception Status and Control field in *cr0*, which causes the hardware to load AIP to IP register. See the STATE_SIP command for how to set SIP.

Although the SIP may be more than 32 bits wide, the EU still only uses the low 32 bits.

Register and Subregister Numbers for Control Register

RegNum[3:0]	SubRegNum[4:0]
0000b = cr0 All other encodings are reserved.	00000b = **cr0.0:ud**. It contains general thread control fields. 00100b = **cr0.1:ud**. It contains exception status and control. 01000b = **cr0.2:ud**. It contains AIP. All other encodings are reserved.

Control Register Fields

DWord	Bits	Description
0	31	**Primary Exception State and Control.** This bit is the primary state and control for all exceptions. Reading a 0 indicates that the thread is in normal operation state and a 1 means the thread is in exception handle state. Upon an exception event, hardware sets this bit to 1 and switches to SIP. Writing 1 to this bit has no effect. Writing 0 to this bit also has no effect if the previous value is 0. In both cases, the bit keeps the previous value. If the previous value of this bit is 1, software writing a 0 causes the thread to return to AIP. This transition is automatic - software does not have to move AIP to IP. The value of this bit then stays as 0. 0 = The thread is in normal state. 1 = The thread is in exception state.

DWord	Bits	Description
		Initialized by hardware. Read/Clear.
	30:16	**Reserved.** MBZ.
	15	**Breakpoint Suppress.** This bit specifies whether breakpoint exception is suppressed or not. This bit is normally set by software and cleared by hardware. If Primary Exception Status and Control bit is 1, this bit is ignored by hardware. If Primary Exception Status and Control bit is 0 (i.e. not in System Routine) and Breakpoint is enabled: If this bit is set, breakpoint is temporally ignored (suppressed); Upon a breakpoint condition, the instruction is executed and this bit is automatically reset by hardware. This bit is provided to prevent infinite loops of jumping to the System Routine on a breakpoint condition. The System Routine must set this bit (and also clear the corresponding status and control bit) before returning to the application thread. This bit has no effect when Breakpoint Enable bits are cleared. This bit is initialized to 0. 0 = Breakpoint exception is not suppressed. 1 = Breakpoint exception is suppressed. Initialized to 0. Read/Set.
	14	**Systolic Mode Enable:** This bit enables systolic operations. If this bit is not set, and a systolic instruction is issued, the Systolic Exception will be triggered. This is to prevent ICC current issues when executing systolic operations, unless Fmax has been lowered first. This bit is initialized on thread dispatch, and cannot be written by software. 0 = Systolic operations are disabled (default) 1 = Systolic operations are enabled Initialized at dispatch. Read-only.
	13	**Large GRF mode.** This field specifies whether the thread is running kernel in Large GRF Mode. 0 = Regular Mode. 1 = Large GRF Mode. Initialized at dispatch. Read-only.
	12	**IEEE Float to Integer Rounding Mode.** This bit determines how rounding modes are handled in float to integer conversion operation. This bit is initialized to 0 during Thread Dispatch. This bit must be set for IEEE compliant float to integer conversion operation. 0= The result of float to integer conversion operation is with RTZ rounding mode. 1= The result of float to integer conversion operation is with rounding mode programmed in cr0.0[5:4]. Initialized to 0. Read/Write.

DWord	Bits	Description
	11	**IEEE MinMax.** This bit determines how SNAN is handled in min/max operations. Refer to IEEE Floating Point Mode for details. This bit is initialized to 0 during Thread Dispatch. This bit must be set for IEEE compliant min/max operation. 0 = The result of min/max is a non-SNAN source. 1 = The result of min/max is the SNAN source. Initialized to 0. Read/Write.
	10	**Half Precision Denorm Mode.** This bit determines how denormal numbers are handled for the HF (Half Float) type. This bit is initialized to 0 during Thread Dispatch. 0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign. 1 = Allow denorm source values and denorm results. Initialized to 0. Read/Write.
	9	**IEEE Exception Trap Enable.** This bit enables trapping IEEE exception flags. This control bit may be updated by software. It is initially zero on thread load. If enabled, IEEE floating-point exceptions set sticky bits in the IEEE Exceptions field of sr0.1. Note that IEEE floating-point exceptions do *not* transfer control to any handler. 0 = IEEE Exception flags are NOT trapped. 1 = IEEE Exception flags are trapped. Initialized to 0. Read/Write.
	7	**Single Precision Denorm Mode.** This bit determines how denormal numbers are handled for the F (Float) type when using the IEEE floating-point mode. It is ignored in the ALT floating-point mode, which always flushs denorms. This bit is initialized by Thread Dispatch. 0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign. 1 = Allow denorm source values and denorm results. Initialized at dispatch. Read/Write.
	6	**Double Precision Denorm Mode.** This bit determines how denormal numbers are handled for the DF (Double Float) type. It is initialized by Thread Dispatch. 0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign. 1 = Allow denorm source values and denorm results. Initialized to 0. Read/Write.

DWord	Bits	Description
	5:4	**Rounding Mode.** This field specifies the FPU rounding mode. It is initialized by Thread Dispatch. 00b = Round to Nearest or Even (RTNE) 01b = Round Up, toward +inf (RU) 10b = Round Down, toward -inf (RD) 11b = Round Toward Zero (RTZ) Initialized at dispatch. Read/Write.
	3	**Vector Mask Enable (VME).** This bit indicates DMask or Vmask should be used by EU for execution. This bit is set by the Thread Dispatch. 0: Use Dispatch Mask (DMASK) 1: Use Vector Mask (VMASK) Initialized at dispatch. Read/Write.
	2	**Single Program Flow (SPF).** Specifies whether the thread has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). This bit affects the operation of all branch instructions. In Single Program Flow mode, all execution channels branch and/or loop identically. This bit is initialized by the Thread Dispatch. 0: Multiple Program Flows 1: Single Program Flow **Programming Restrictions:** Only H1/Q1/N1 are allowed in SPF mode. Power Optimization: If an entire shader does not do SIMD branching, the driver can set the SPF bit to 1 to save power in HW. SPF mode must be set to 0 in Fused threads. Initialized at dispatch. Read/Write.
	1	**Accumulator Disable.** This bit controls the update of the accumulator by the instruction field AccWrCtrl. If this bit is cleared, the accumulator is updated for all instructions with AccWrCtrl enabled. If set, the accumulator is disabled for all update operations, maintaining its value prior to being disabled. Setting this bit has no effect if the accumulator is the explicit destination operand for an instruction. This bit is initialized to 0. 0: Enable accumulator update. 1: Disable accumulator update. **Usage Notes:** This control bit is primarily designed for the System Routine. That routine is not expected to use the accumulator, though it may need to use instructions that implicitly update the accumulator. To use such instructions in the System Routine, but still preserve the accumulator contents on

DWord	Bits	Description
		returning to the application kernel, the System Routine would either (a) save and restore the accumulator, or (b) prevent the accumulator from being unintentionally modified. This control bit has been added for the latter method. Software has the option to limit the setting of this control bit to strictly within the System Routine. If, by convention, this bit is clear within application kernels, the System Routine can simply set the bit upon entry and clear it before returning control to the application kernel. This usage model would not necessarily require cr0.0 to be saved/restored in the System Routine. However, if by convention application kernels are permitted to set this bit, then the System Routine is required to preserve the content of this bit. Initialized to 0. Read/Write.
	0	**Single Precision Floating Point Mode (FP Mode).** This bit specifies whether the current single-precision floating-point operation mode is IEEE mode (IEEE Standard 754) or the ALT (alternative mode). This bit does not affect the floating-point mode used for other floating-point data types. Refer to Alternative Floating Point Mode for details. This bit is also forwarded on the message sideband for all out-going messages, for example, to control the floating-point mode of the Sampler. Software may modify this bit to dynamically switch between the two floating-point modes. This bit is initialized by Thread Dispatch. 0 = IEEE floating-point mode for the F (Float) type. 1 = ALT (alternative) floating-point mode for the F (Float) type. Initialized at dispatch. Read/Write.
1	31	**Breakpoint Exception Status and Control**. This bit, when set, indicates a breakpoint exception. Under breakpoint condition, hardware sets this bit upon entering the System Routine. For normal breakpoint handling, the system routine should reset this bit and also set the Breakpoint Suppress bit before returning back to the application thread. If this bit is not reset in the System Routine, upon returning to an application routine, hardware executes one instruction (if Breakpoint Suppress bit was set in the System Routine), and then jumps to the System Routine again. Thus, by not clearing this bit, single stepping can be emulated. This bit may be set and cleared by software. This bit is initialized to 0. Initialized to 0. Read/Write.
	30	**External Halt Exception Status and Control**. This bit indicates the External Halt exception. It is set by EU hardware on receiving the broadcast External Halt signal. The System Routine should reset this bit before returning to an application routine to avoid infinite loops. This bit may be set or cleared by software. This bit is initialized to 0. Initialized by hardware. Read/Clear.

DWord	Bits	Description
	29	**Software Exception Control**. This bit is the control bit for software exceptions. Setting this bit to 1 in an application routine causes an exception. Clearing this bit in an application routine has no effect. Upon entering the system routine, the hardware maintains this bit as 1 to signify a software exception. The System Routine should reset this bit before returning to an application routine. This bit may be set or cleared by software. This bit is initialized to 0. Initialized to 0. Read/Write.
	28	**Illegal Opcode Exception Status**. This bit, when set, indicates an illegal opcode exception. The exception handler routine normally does not return back to the application thread upon an illegal opcode exception. Leaving this bit set has no effect on hardware; if system software adversely returns to an application routine leaving this bit set, it doesn't cause any exception. This bit should not be set by software or left set by the system routine to avoid confusion. Initialized to 0. Read/Clear.
	27	**Systolic Exception Status.** This bit, when set, indicates a systolic exception. The exception handler routine normally does not return back to the application thread upon an illegal opcode exception. Leaving this bit set has no effect on hardware; if system software adversely returns to an application routine leaving this bit set, it doesn't cause any exception. This bit should not be set by software or left set by the system routine to avoid confusion. This bit is initialized to 0.
	25	**Context Save Status.** This bit when set, indicates a Context Save process has been initiated. The system routine must reset this bit after saving the context to terminate the thread.
	24	**Context Restore Status.** This bit when set, indicates a Context Restore process has been initiated. The system routine must reset this bit after restoring the context. The reset of this bit is required before invoking application routine.
	23:16	**Reserved**. MBZ.
	13	**Software Exception Enable**. This bit enables or disables the software exception. Enabling or disabling this bit may allow host software to turn on/off certain features (such as profiling) without changing the kernel program. This bit is initialized by the Thread Dispatcher. Format = ENABLED: 0: Disabled 1: Enabled Initialized at dispatch. Read/Write.

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td></td><td>12</td><td>Illegal Opcode Exception Enable. This bit specifies whether the illegal opcode exception is enabled or not. The Illegal opcode exception includes illegal opcodes and undefined opcodes, caused by bad programs or run-time data corruption.

This bit is initialized by the Thread Dispatcher.

Software should normally assign this bit in the interface descriptor. Even though this mechanism is provided to disable the illegal opcode exception, it should be used with extreme caution.

Format = ENABLED:

0: Disabled

1: Enabled

Initialized at dispatch. Read/Write.</td></tr>
<tr><td></td><td>11:0</td><td>Reserved. MBZ.</td></tr>
<tr><td>2

(cr0.2:ud)</td><td>31:3</td><td>Application IP (AIP). This is the register storing the instruction pointer before an exception is handled. Upon an exception, hardware automatically saves the current IP into the AIP register, and then sets the Primary Exception State and Control field to 1, which forces a switch to the System IP (SIP). The AIP register may contain either the pointer to the instruction that causes the exception (such as breakpoint) or the one after (such as masked stack overflow/underflow exceptions). This is shown in the following table, where IP is the instruction that generated the exception.

<table><tr><th>Exception Type</th><th>AIP Value</th></tr><tr><td>External Halt</td><td>N/A (1)</td></tr><tr><td>Software Exception</td><td>IP + 1</td></tr><tr><td>Illegal Opcode</td><td>IP</td></tr></table>
(1) External Halt exception is asynchronous and not associated with an instruction.

When the System Routine changes the Primary Exception State and Control field from 1 to 0, hardware restores IP from this register. This field is writable allowing the returning IP to be altered after an exception is handled.

Format = U32[31:3]

Not Initialized. Read/Write.</td></tr>
<tr><td></td><td>2:0</td><td>Reserved. MBZ.</td></tr>
</table>

Implementation Restriction on Register Access: When the control register is used as an explicit source and/or destination, hardware does not ensure execution pipeline coherency. Software must make the immediate next instruction dependent on an instruction that uses control register as an explicit operand. This is important as the control register is an implicit source for most instructions. For example, fields like FPMode and Accumulator Disable control the arithmetic and/or logic instructions. Therefore, if the instruction updating the control register doesn't make the next instruction stall, subsequent instructions may have undefined results.

Notification Registers

Notification Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1001b
Number of Registers:	1
Default Value:	No
Normal Access:	RO/PW
Elements:	2
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control?	No
Indexable?	No

Register and Subregister Numbers for Notification Registers

RegNum[3:0]	SubRegNum[4:0]
0000b = n0 All other encodings are reserved.	00000b = **n0.0:ud** 00100b = **n0.1:ud** 01000b = **n0.2:ud** All other encodings are reserved.

There are three notification registers (*n0.0:ud*, *n0.1:ud, and n0.2:ud*) used by the *wait* instruction. These registers are read-only, except under context restore, and can be accessed in 32-bit granularity. Write access to this register is allowed only when context is restored.

Programming Note	
Context:	Notification Registers - Workaround
The sub-register numbers for n0.0 and n0.2 are swapped on a write, i.e., a destination of n0.0 is required to update n0.2 and n0.2 is required to update n0.0.	

It should be noted that in the extreme case, it is possible to have more notifications to a thread than the maximum allowed number of concurrent threads in the system. Therefore, the range of the thread-to-thread notification count in n0, is larger than the maximum number of threads computed by EUID * TID.

There is only one bit for the host-to-thread notification count in n1.

When directly accessed, this register is read-only. If the value is nonzero, the only way to alter the value is to use the wait instruction to decrement the value until zero is reached. A wait instruction on a zero

notification subregister causes the thread to stall, waiting for a notification signal from outside targeting the same subregister. See the wait instruction for details.

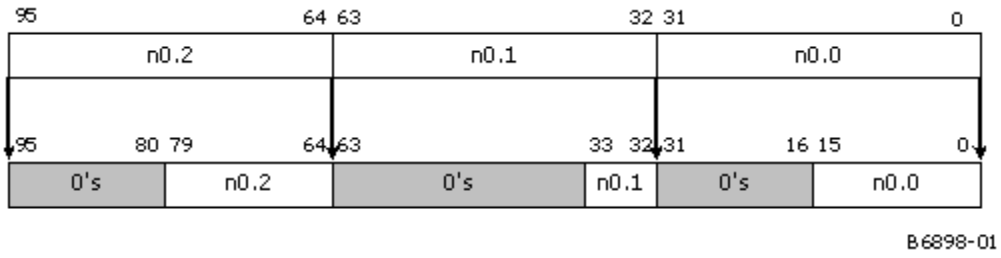
Notification Register 0 Fields

Dword	Bits	Description
0	31:1	**Reserved**. MBZ.
	0	**Thread to Thread Notification Count**. This register is used by the sync.bar instruction for thread-to-thread synchronization. The value read from this register specifies the outstanding notifications received from other threads. It can be changed indirectly by using the sync.bar instruction. See the sync.bar instruction for details. Format: U1 Initialized to 0. Read/PWrite.

Notification Register 1 Fields

DWord	Bits	Description
0	31:1	**Reserved**. MBZ.
	0	**Host to Thread Notification**. This register is used by the sync.host instruction for host-to-thread synchronization via MMIO registers. Format: U1 Initialized to 0. Read/PWrite.

Format of the Notification Register



IP Register

IP Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1010b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	1

Attribute	Value
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control?	No
Indexable?	No

The ip register can be accessed as a 32-bit quantity. It is a read-write register, containing the current instruction pointer, which is relative to the **Instruction Base Address**. Reading this register returns the instruction pointer of the current instruction. The 3 LSBs are always read as zero. Writing this register causes program flow to jump to the new address. Writes to this register should use the Switch ThreadCtrl option. When it is written, the 3 LSBs are dropped by hardware.

Register and Subregister Numbers for IP Register

RegNum[3:0]	SubRegNum[4:0]
0000b = ip All other encodings are reserved.	00000b = **ip:ud** All other encodings are reserved.

IP Register Fields

DWord	Bits	Subfield Description
0	31:3	**Ip**. Specifies the current instruction pointer. This pointer is relative to the **Instruction Base Address**.
	2:0	**Reserved**. MBZ.

Performance Registers

Performance Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1100b
Number of Registers:	1
Default Value:	0h
Normal Access:	RO/RW
Elements:	5
Element Size:	32 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control?	No
Indexable?	No

Timestamp Register

This register is a low latency timestamp source, "TM", available as part of a thread's Architectural Register File (ARF). This is a is free running counter, 64b in size, and exposed to the ISA as individual 32b high 'TmHigh' and low 'TmLow' unsigned integer source operands. As part of the EU's register space, access to the timestamp has a low and deterministic latency and therefore can be used for intra-kernel high resolution performance profiling.

The TM features provides a 1-bit indicator 'TmEvent' which identifies the occurrence of a time-impacting event such as context switch or frequency change since the last time any portion of the Timestamp register value was read by that thread. Software that uses the Timestamp capability should check this bit to identify when a relative time calculation may be suspect. To properly use this additional information, the instrumentation code should operate on the Timestamp register value as a whole (i.e. as an 8 dword register) so that the 64b time and this 1b value are captured simultaneously, as opposed to 32b portions, to eliminate the chance of missing a TmEvent that might occur between accesses to 32b portions of this register.

Programming Note	
Context:	Performance Registers
The Timestamp register need not be saved as part of thread state on context-save, but only 'TmEvent' must be set to 1 as part of context restored to indicate that context switch had occurred).	

Performance Counter Register

This is a counter intended to provide finer grained visibility into the EUs performance inside kernels. This counter is a 32-bit free-running counter that increments if the EU flexible performance event selected for OA counter A7 is true (please refer to OA documentation for details on how to count various EU flexible events on OA counter A7). The pm0 count continues to increment during a thread's active/standby state transitions as well as context switches. It is read-only and not pre- or resettable under any software control, either kernel or driver, other than a full gfx reset.

Pause Register

This register provides the mechanism for a thread to pause itself from further execution for a short amount of time. This may be useful in situations where a periodic polling operation on an external resource is required, but polling loop time needs to be controlled to prevent excessive consumption of execution slots and resource bandwidth. To mitigate excessive polling rates, this 'pause' operation can be placed in the polling loop to cap the periodic polling at some maximum rate.

The 'pause' operation is invoked by the kernel itself through the writing of the Pause Register with an unsigned-word value. The register over the course of many clocks will count down from the written value to zero, in steps of 32 decrements every 32 EU clocks (first decrement event has an uncertainty between 1 to 32 clocks). Upon reaching 0x0, the decrement stops. During the time the Pause Register is non-zero, no new instruction issue occurs; upon reaching a 0-count, the thread once again becomes a candidate for execution. Note that any instruction or message issued prior to the invocation of the 'pause' continues to execute and retire. Actual resumption of instruction issues after the pause duration may further be delayed through normal operational policies such as thread priority and/or outstanding register dependencies.

Therefore, the value written for 'pause' should be considered a number of instruction issue slots (divide by 2 for SIMD-8 instruction slots, or 4 for SIMD-16 instruction slots), as opposed to some fixed time duration.

The Pause Register is reset to 0x0 when a new thread is loaded. It is also reset to 0x0 upon invocation of the System Thread IP (SIP) and at the commencement of a context save or context restore event. The value of the register is not saved/restored as part of context save/restore.

The actual duration of the pause is considered approximate; generally, the duration will be somewhat longer than the value written, as the counting does don't commence until the write to the 'Pause' register actually retires.

Register and Subregister Numbers for Performance Register

RegNum[3:0]	SubRegNum[4:0]
0000b = *tm0* All other encodings are reserved.	00000b = **tm0.0:ud.** 00100b = **tm0.1:ud.** 01000b = **tm0.2:ud** 01100b = **tm0.3:ud** 10000b = **tm0.4:ud** **All other encodings are reserved.**

Performance Register Fields

DWord	Bits	Description
0 (tm0.0:ud)	31:0	**TmLow.** The lower 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32
1 tm0.1:ud	31:0	**TmHigh.** The upper 32b of the 64b timestamp value sourced from Cr clock. Read-only. Format: U32
2 tm0.2:ud	31:1	**Reserved**
	0	**TmEvent.** Indicates a discontinuous time-impacting event (e.g. context switch, frequency change) occurred since any portion of the Timestamp register was last read, thus making any relative duration calculation based on this counter suspect. This bit is reset at the time a new thread is loaded, and on each read of any portion of the 'Timestamp' register.
3 tm0.3 (pm0)	31:0	pm0. Increments based on the EU flexible performance event currently selected being true. Format: U32
	31:16	**Reserved**

DWord	Bits	Description
4 tm0.4:ud (tp0)	15:0	**Pause Counter.** The pause duration. A non-zero value written to this register causes execution of the thread to halt for the corresponding number of clocks. Lower 5 bits are always zero and therefore, writing value less than 64 may not result in a pause. [15:10] - Reserved, must be written as zero; when read, returns zero. [9:5] - Count value. [4:0] - Reserved, must be zero. Format: U16

Flow Control Registers

Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	1101b
Number of Registers:	4
Default Value:	None
Normal Access:	R/PW
Elements:	32
Element Size:	64 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control?	No
Indexable?	No

Register and Subregister Numbers for Flow Control Registers

RegNum[3:0]	SubRegNum[4:0]
0000b = fc0	00000b-11111b = **fc0.0-fc0.31**.
0001b = fc1	00000b = **fc1.0**. All other encodings are reserved.
0010b = fc2	00000b = **fc2.0**. All other encodings are reserved.
0011b = fc3	00000b = **fc3.0**. All other encodings are reserved.

These are special hardware registers used in handling flow control operations. These registers may be accessed ONLY in context save/restore operation using the SIP. These registers are accessed with the 'MOV' opcode. Use of any other opcode or access of these registers in non-context save/restore modes may result in indeterministic behavior of hardware.

These registers are accessed as 256b registers. Parts of the 256b register may be redundant, depending on the hardware implementation of each register. The fields "RegNum" and "SubRegNum" are used together to address these registers.

Immediate

Two forms of immediate are provided as a source operand: scalar and vector.

The immediate field may be 64 bits or 32 bits. For a word, unsigned word, or half-float immediate data, software must replicate the same 16-bit immediate value to both the lower word and the high word of the 32-bit immediate field in an instruction. The 64-bit immediate takes up two DWords of the instruction bit field. Hence a 64-bit immediate is supported ONLY for a MOV operation. The field is denoted by ***imm32:type*** for 32-bit immediates and ***imm64:type*** for 64-bit immediates.

The immediate form of vector allows a constant vector to be in-lined in the instruction stream. Both integer and float immediate vectors are supported.

An immediate integer vector is denoted by type **v** or **uv** as *imm32:v* or *imm32:uv*, where the 32-bit immediate field is partitioned into 8 4-bit subfields. Refer to the Numeric DataType topic for description of the packing of vector integers to a DWord.

For a scalar immediate, the numeric data types supported are ***:uw, :w, :ud, :d, :uq, :q*** for integers AND ***:hf, :f, :df*** for floats. Refer to the Instruction Machine format topics for the encoding of these immediates.

An immediate float vector is denoted by type **vf** as *imm32:vf*, where the 32-bit immediate field is partitioned into 4 8-bit subfields. Refer to the Numeric DataType topic for the description of the packing of vector floats to a DWord.

When an immediate vector is used in an instruction, the destination must be 128-bit aligned with destination horizontal stride equivalent to a word for an immediate integer vector (***v***) and equivalent to a DWord for an immediate float vector (***vf***).

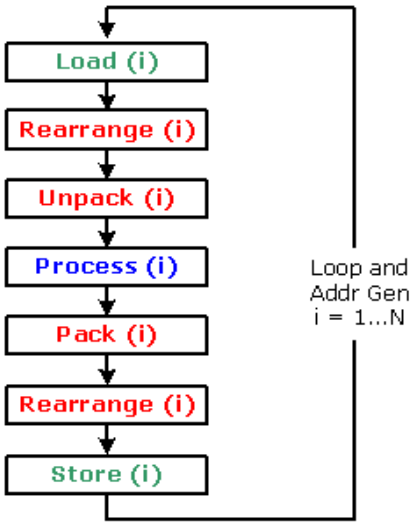
Region Parameters

Unlike conventional SIMD architectures where an N-bit wide SIMD instruction can only operate on N-bit aligned SIMD data registers, a region-based register addressing scheme is employed in architecture. The region-based register addressing capability significantly improves the SIMD computation efficiency by providing per-instruction-based multiple data gathering from register file. This avoids instruction overhead to perform data pack, unpack, and shuffling, which has been observed on other SIMD architectures. One benefit of such capability is allowing various kinds of 3D Graphics API Shader compute models to run efficiently on GFX. Another benefit is allowing high throughput of media applications, which tend to operate on byte or word data elements.

This can be illustrated by the example shown in Conventional SIMD Instruction Sequence and SIMD Instruction Sequence for the Same Program. As shown in Conventional SIMD Instruction Sequence, a sequence of SIMD instruction is executed on a conventional load/store-based superscalar machine with SIMD instruction extension. The data parallelism can be achieved by first level of loop unrolling. As shown, there is a second level of loop for the task. Before a given SIMD compute instruction, *Process (i)*, can proceed, there might be a load, a data rearrange, and a data unpack (and conversion) instruction to

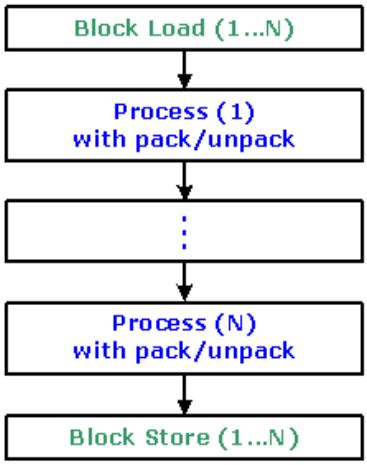
load and prepare the input data. After the compute instruction is complete, it might also require pack, re-arrange and store instructions, to format and save the same to memory. At the loop, other scalar computations such as loop count and address generation may be needed. For the same program, when the data can fit in the large GRF register file, the outer loop may be unrolled for GFX. Here one or a few block loads (using *send* instruction) may be sufficient to move the working set into GRF. Then the data shuffle can be combined with the processing operation with region-based addressing capability. Per operand float type and mixed data type operation may also allow GFX to combine data conditioning operations with computing operations. These techniques in architecture help to achieve high compute efficiency and throughput for graphics and media applications.

Conventional SIMD Instruction Sequence



B6899-01

SIMD Instruction Sequence for the Same Program



B6900-01

In an instruction, each operand defines a region in the register file. A region may contain multiple data elements. Each data element is assigned to an execution channel in the EU. The total number of data elements of a region is called the **size** of the region, or the size of the operand. The number of execution channels is called the **execution size** (*ExecSize*), which is specified in the instruction word. ExecSize determines the size of region for source and destination operands in an instruction.

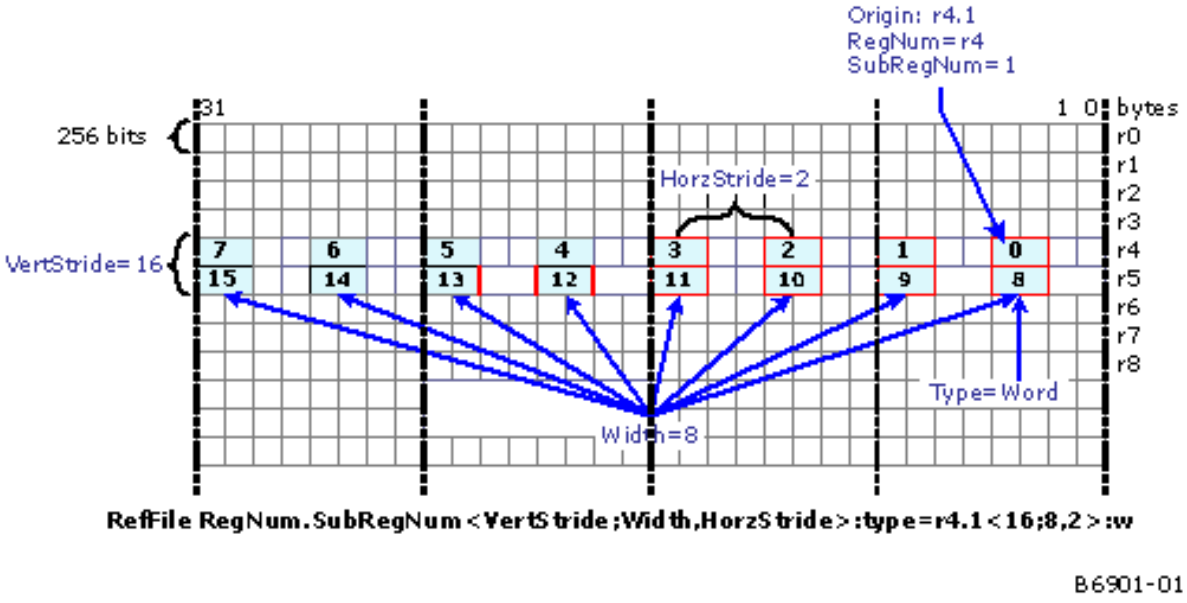
- For an instruction with two source operands, the sizes of the two source operands must be the same.
- The size of a destination operand generally matches the execution size, therefore equals to the number of source operand(s) in the same instruction.
    - Exception of this rule is present for the integer reduction instructions (such as sad2 and sada2) where the destination area is smaller than the source area.

Regions are **generalized 2-dimensional** (2D) arrays in row-major order. The first dimension is named the **horizontal** dimension (data elements within a row) and the second dimension is termed the **vertical** dimension (data elements in a column). Here, horizontal/vertical and row/column are just symbolic notations.

When the GRF registers are viewed as a row-major 2D array of memory, such a notation normally matches well with the geometric locations of the data elements of an operand.

However, as the register region is fully described by the parameters discussed below, the data elements of a register region may not form a regular rectangular shape. For example, Vertical Stride parameter is allowed to be smaller than Horizontal Stride, making the rows of a register region interleave with each other. It should also note that the meanings of horizontal/vertical here is different than that used for the flag control in Section Flag Register

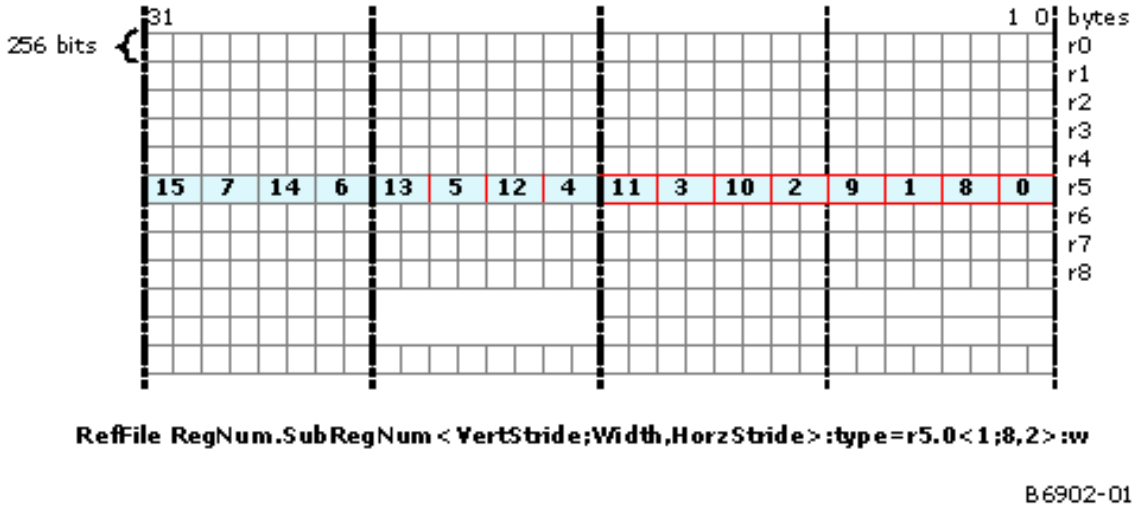
An example of a register region (*r4.1<16;8,2>:w*) with 16 elements



A 16-element register region with interleaved rows (r5.0<1;8,2>:w) shows another example where the rows are interleaved. The region, having word data elements, starts at location r5.0:w. HorzStride, the distance within a row, is 2 words. So the second element (channel number 1) is at location 5.2:w. And there are 8 elements per row. VertStride, the distance between two rows, is only 1 word, which is less than HorzStride. Therefore, the first element of the second row (channel number 8) is at r5.1:w, just next to channel number 0. It is clear from the picture that the two rows are interleaved.

By varying the region parameters, reader may construct other configurations. The next section provides more details on the region-based register addressing. However, there are restrictions imposed by hardware implementation, which can be found in the later sections of this chapter.

A 16-element register region with interleaved rows (*r5.0<1;8,2>:w*)



Without considering the source channel swizzle and destination register region description, the above row-major-order region description provides the data assignment to each execution channel. The following pseudo code computes the addresses of data elements assigned to execution channels for a special case when the destination register is aligned to 256-bit register boundary.

```
// Input: Type: ub | b | uw | w | ud | d | f | v
//RegNum: In unit of 256-bit register
//SubRegNum: In unit of data element size
//ExecSize, Width, VertStride, HorzStride: In unit of data elements
// Output: Address[0:ExecSize-1] for execution channels
int ElementSize = (Type=="b"||Type=="ub") ? 1 : (Type=="w"|Type=="uw") ? 2 : 4;
int Height = ExecSize / Width;
int Channel = 0;
int RowBase = RegNum«5 + SubRegNum * ElementSize;
for (int y=0; y<Height; y++) {
        int Offset = RowBase;
```

```
        for (int x=0; x<Width; x++) {
    Address [Channel++] = Offset;
    Offset += HorzStride*ElementSize;
        }
        RowBase += VertStride * ElementSize;
}
```

The following pseudo code computes the addresses of data elements assigned to execution channels for a special case when the destination register is aligned to 512-bit register boundary.

```
// Input: Type: ub | b | uw | w | ud | d | f | v
//RegNum: In unit of 512-bit register
//SubRegNum: In unit of data element size
//ExecSize, Width, VertStride, HorzStride: In unit of data elements
// Output: Address[0:ExecSize-1] for execution channels
int ElementSize = (Type=="b"||Type=="ub") ? 1 : (Type=="w"|Type=="uw") ? 2 : 4;
int Height = ExecSize / Width;
int Channel = 0;
int RowBase = RegNum«6 + SubRegNum * ElementSize;
for (int y=0; y<Height; y++) {
   int Offset = RowBase;
      for (int x=0; x<Width; x++) {
Address [Channel++] = Offset;
Offset += HorzStride*ElementSize;
      }
      RowBase += VertStride * ElementSize;
}
```

As HorzStride and VertStride are specified independently (note that VertStride might be smaller than or equal to HorzStride), the region may take various shapes from a replicated scalar, a replicated vector, a vector of replicated scalars, a sliding window, to a non-overlapped 2D array.

A region-based description of a destination operand can take the following simplified format

RegFile RegNum.SubRegNum<HorzStride>:type

The destination operand is only allowed to have a 1 dimensional region. The Register Region Origin and Type are the same as for a source operand. The total number of elements is given by ExecSize. However, only HorzStride is required to describe the 1D region, not VertStride and Width.

As a source register region may cross multiple physical GRF registers, an instruction with such source operands may take more than two execution cycles to gather source data elements for execution. The destination register region of a non-compressed instruction is restricted to be within a physical GRF register. In other words, destination scatter writes over multiple registers are not supported.

Region Addressing Modes

There are two different register addressing modes: Direct register addressing and register-indirect register addressing. Depending on the register region description, the register-indirect register addressing mode can be further divided into three usages: 1x1 index region where only the origin of register region is provided by the address register, Vx1 index region where the offset of each row of the register region is provided by an address register, VxH index region where the offset of each data element is provided by an address register.

The 16-bit address register has byte address of the GRF register, additionally an immediate byte address offset can also be specified in the instruction. The sum of the address register and immediate offset can be divided by GRF register size to determine the RegNum and SubRegNum.

The examples in this section use 256-bit physical GRF registers to demonstrate the concept.

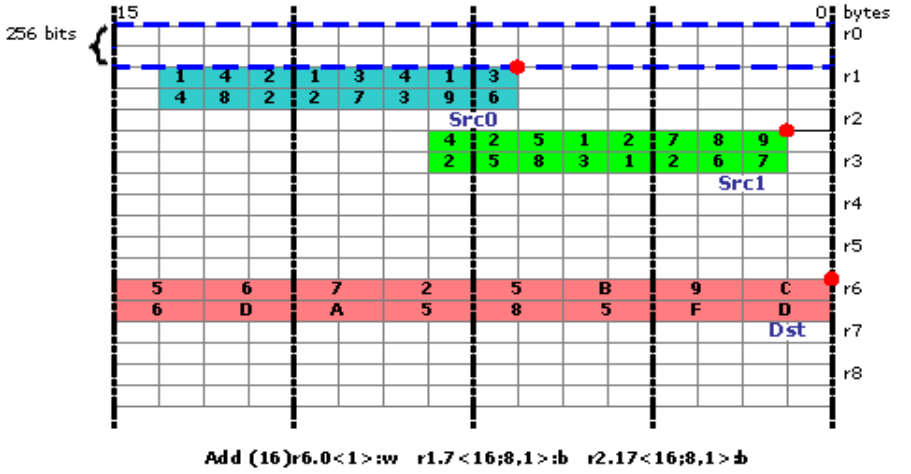
Direct Register Addressing

In this mode, all register region parameters are specified for an operand using fields in the instruction word.

Direct Register Addressing and *Direct Register Addressing* are two examples of direct register addressing.

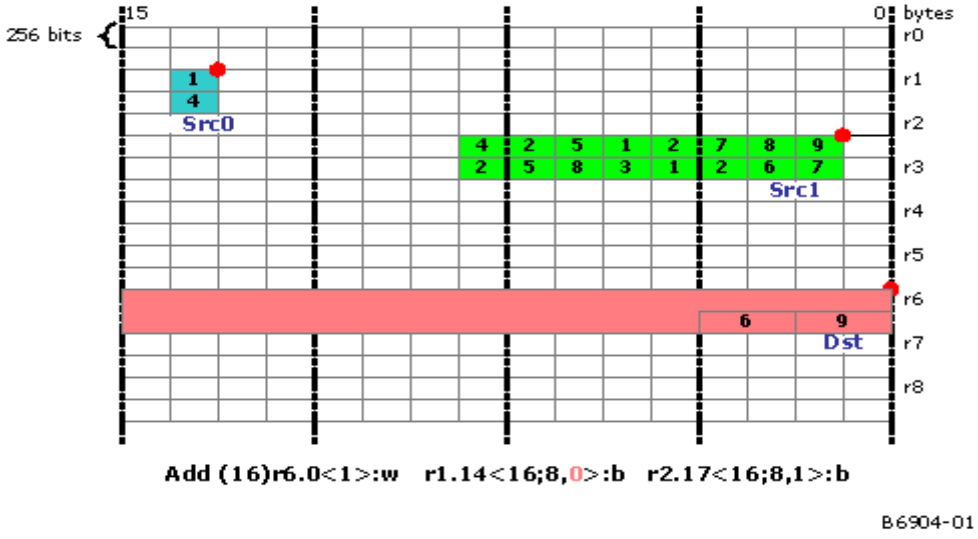
For the example in *Direct Register Addressing*, all operands are 2D rectangular regions having the same size of 16 data elements. The two source operands, *Src0* and *Src1*, have 16 bytes. The destination operand, *Dst*, has 16 words. There are 8 elements in a row for *Src0* and *Src1*. The vertical stride of 16 bytes for *Src0* and *Src1* indicates that the first element and the 9'th element are 16 bytes apart in the register file. Note that *Src0* falls into the 256-bit physical GRF register starting at r1.0, but Src1 crosses the 256-bit physical GRF register boundary between r2 and r3. The numbers in the shaded regions are the values of the data elements. Observing the upper right corners of the source/destination regions (first data element), we have C = 3+9.

A region description example in direct register addressing



For the example in *Direct Register Addressing*, the sizes of areas of *Src0* and *Src1* are the same, but *Src0* contains a vector of replicated scalars. With HorzStride = 0 and Width = 8, the first row of 8 elements in Src0 is a replication of the byte at r1.14. Comparing *ExecSize* of 16 to Width of 8 indicates that there is a second row of 8 elements in *Src0*. With VertStride = 16, the second row in *Src0* is a replication of the byte at r1.30 (30 = 14+16). Effectively, the 16 data elements of *Src0* are {1,1,1,1,1,1,1,1, 4,4,4,4,4,4,4,4}.

A region description example in direct register addressing with src0 as a vector of replicated scalars

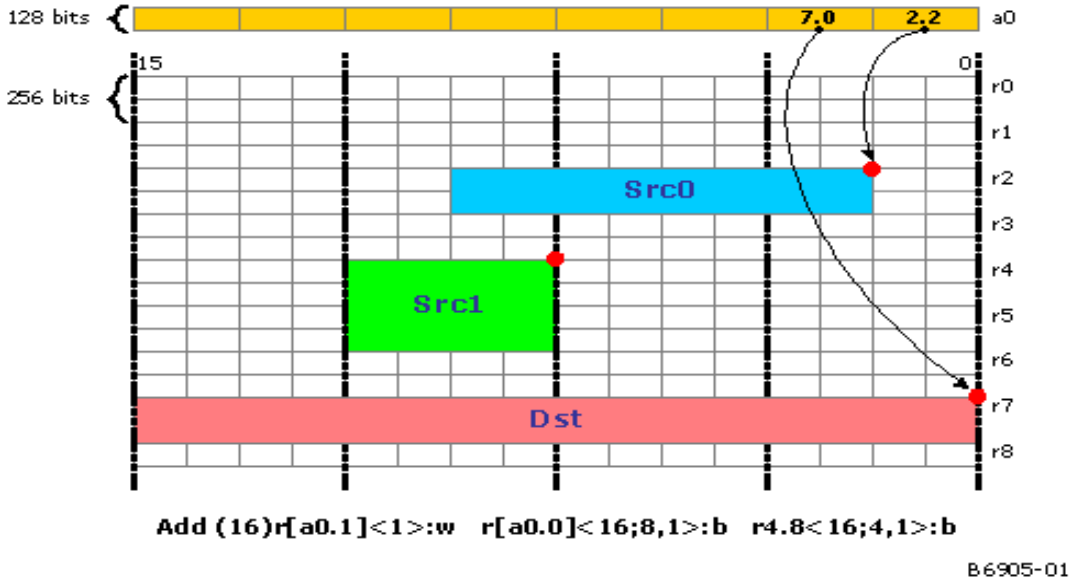


Register-Indirect Register Addressing with a 1x1 Index Region

In the register-indirect register addressing mode with 1x1 index region, the region origin is provided by the content of the address register, the rest of region parameters are provided by the fields in the instruction word.

Register-Indirect Register Addressing with a 1x1 Index Region depicts an example for this addressing mode. For example, the presence of a full region description <16;8,1> for Src0 indicates that only the origin of the region is provided by the address register a0.0.

An example illustrating register-indirect register addressing mode with a 1x1 index region

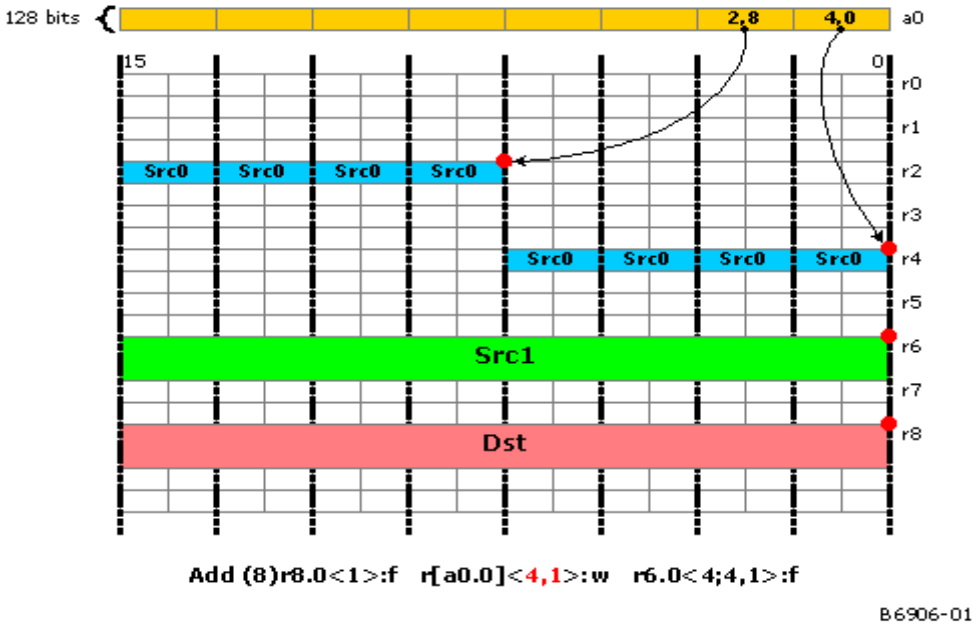


Register-Indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, the horizontal dimension is described by the fields in the instruction word and the vertical dimension is described by an address register region. Specifically, the origin of each row of the data region is provided by the contents of an address register region. The rows are described by the width and the horizontal stride. The first address register is provided and the following contiguous address registers are for the following rows. The total number of address registers used is inferred from the parameters *ExecSize* and *Width*.

An example is provided in *Register-Indirect Register Addressing with a Vx1 Index Region*. The assembly syntax notion of a register region without vertical stride, <4,1>, corresponding to the special encoding of vertical stride of 0xF in the instruction word, indicates the VxH or Vx1 mode of indirect register addressing. In this case, the origin for each row of src0 is provided by the address register. As ExecSize/Width = 2, there are two address registers a0.0 and a0.1, each pointing to a row of 4 data elements.

An example illustrating register-indirect-register addressing mode with a Vx1 index region (src0)

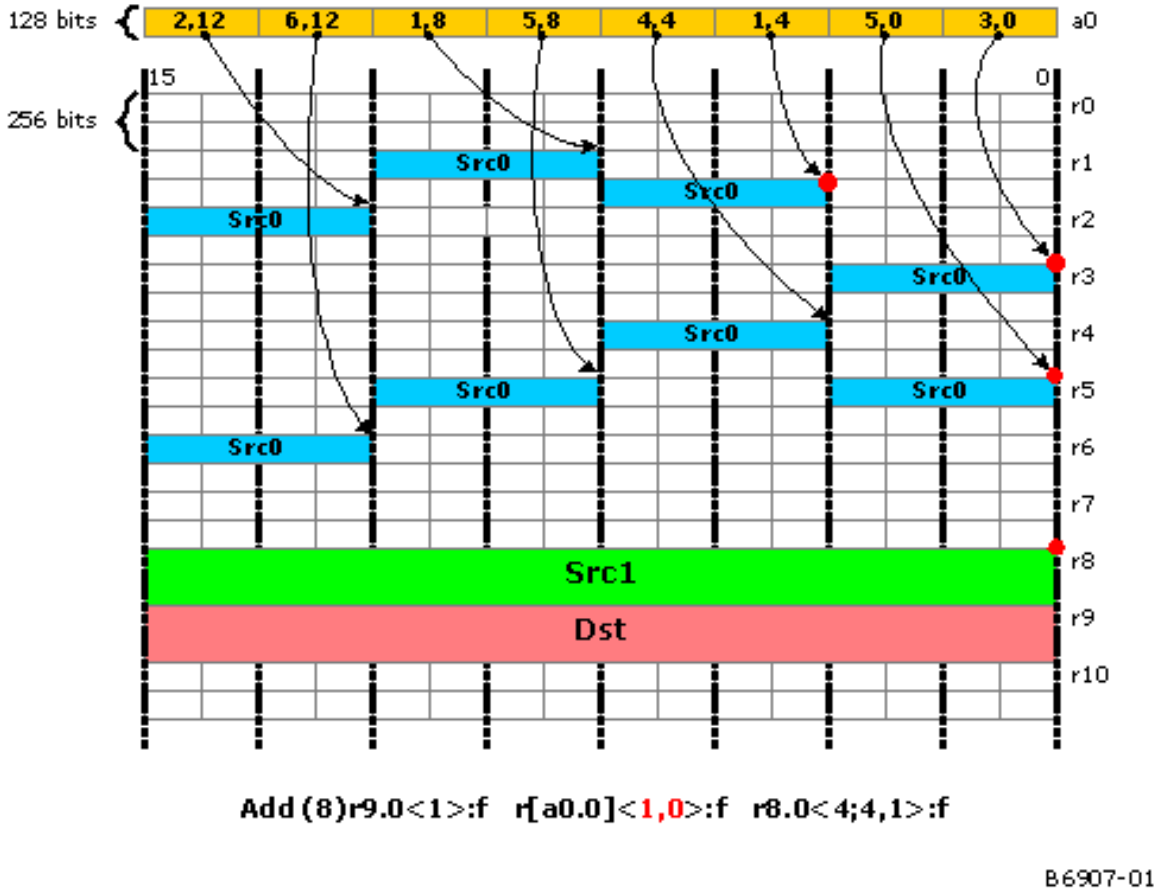


Register-Indirect Register Addressing with a VxH Index Region

In the register-indirect register addressing mode with VxH index region, the position of each data element is provided by the contexts in an address register region. This mode has the identical syntax as the Vx1 index region mode, and in fact, can be viewed as a special case of the Vx1 mode. When *Width* of the region is 1, the number of address registers used equals *ExecSize*.

An example is provided in *Register-Indirect Register Addressing with a VxH Index Region*. The absent of vertical stride in the region description <1,0> with width = 1 indicates that the origin for each row of 1 data element of Src0 is provided by the address register. As ExecSize/Width = 8, there are 8 address registers from a0.0 to a0.7, each pointing to a single data elements.

An example illustrating register-indirect register addressing mode with a VxH index region (Src0).



Access Modes

There are two basic register access modes controlled by a single bit instruction subfield called Access Mode.

- 16-byte Aligned Access Mode (**align16**): In this mode, the origins of all operands (sources and destination), whether it is by direct addressing or register-indirect addressing, are 16-byte aligned. For example, a row in the region description starts at 16-bype aligned and the width the row must be 4 and the 4 data elements within a row must span 16-bytes. In this access mode (and with other restrictions put forward later), full-channel swizzle for both source operands and per-channel mask for destination operand are supported on a 4-component basis. In other words, the control and setting of full source swizzle and destination mask are repeated for every 4 components up to total of *ExecSize* channels.
    - The **align16** access mode can be used for AOS operations. See examples provided in the Primary Usage Model section for SIMD4x2 and SIMD4x1 modes of operation to support 3D API Vertex Shader and Geometric Shader execution.
- 1-byte Aligned Access Mode (**align1**): In this mode, the origins of all operands may be aligned to their data type and could be 1-byte if the operand is of byte type. In this access mode, full region

register descriptions are supported, however, source swizzle or destination mask are not supported.

- The **align1** access mode can be used for SOA operations. See examples provided in the Primary Usage Model section for SIMD8 and SIMD16 modes of operation to support 3D API Pixel Shader. Many media applications also operate well in **align1** access mode.

- Align16 accessmode is restricted to IEEE macro instructions only. All other non-IEEE macro instructions must use Align1 accessmode.

Execution Data Type

The architecture carries out arithmetic and logical operations using a smaller set of data types than the variety supported as source or destination operands. These are the *execution data types*. A particular arithmetic or logical instruction has one execution data type, from those listed in the table.

Execution Data Types

Type	Description
W	Word. 16-bit signed integer.
D	Doubleword. 32-bit signed integer.
Q	Quadword. 64-bit signed integer.
F	Float. 32-bit single precision floating-point number.
DF	Double Float. 64-bit double precision floating-point number.
HF	Half Float. 16-bit half precision floating-point number.

The following rules explain the conversion of multiple source operand types, possibly a mix of different types, to one common execution type:

- For floating-point sources, all source operands must have the same floating-point type, with the exceptions below:
  - A two-source floating-point instruction can have Float as the src0 type and VF (Packed Restricted Float Vector) as the immediate src1 type.
- Mixing floating-point and integer source types is not allowed. Either all source types must be one floating-point type or all source types must be integer types.
- Unsigned integers are converted to signed integers.
- Byte (B) or Unsigned Byte (UB) values are converted to a Word or wider integer execution type.
- If source operands have different integer widths, use the widest width specified to choose the signed integer execution type.

Note that when the execution data type is an integer type, it is always a signed integer type. For integer execution types, extra precision is provided within the hardware, including the accumulators, so that conversions from unsigned to signed do not affect instruction correctness.

Register Region Restrictions

A register region is described as *packed* if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist.

The following register region rules apply to ALU instructions:

1. **General Restrictions Based on Operand Types**

   There are these general restrictions based on operand types:

   1. Where *n* is the largest element size in bytes for any source or destination operand type, *ExecSize* * *n* must be <= 64.
   2. When the Execution Data Type is wider than the destination data type, the destination must be aligned as required by the wider execution data type and specify a *HorzStride* equal to the ratio in sizes of the two data types. For example, a *mov* with a D source and B destination must use a 4-byte aligned destination and a *Dst.HorzStride* of 4.

2. **General Restrictions on Regioning Parameters**

   The mapping of data elements within the region of a source operand is in row-major order and is determined by the region description of the source operand, the destination operand, and the *ExecSize*, with these restrictions:

   1. *ExecSize* must be greater than or equal to *Width*.
   2. If *ExecSize* = *Width* and *HorzStride* != 0, *VertStride* must be set to *Width* * *HorzStride*.
   3. If *ExecSize* = *Width* and *HorzStride* = 0, there is no restriction on *VertStride*.
   4. If *Width* = 1, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.
   5. If *ExecSize* = *Width* = 1, both *VertStride* and *HorzStride* must be 0.
   6. If *VertStride* = *HorzStride* = 0, *Width* must be 1 regardless of the value of *ExecSize*.
   7. *Dst.HorzStride* must not be 0.
   8. *VertStride* must be used to cross GRF register boundaries. This rule implies that elements within a '*Width*' cannot cross GRF boundaries.

3. **Region Alignment Rules for Direct Register Addressing**
   1. In Direct Addressing mode, a source cannot span more than 2 adjacent GRF registers.
   2. A destination cannot span more than 2 adjacent GRF registers.
   3. When a source or destination spans two registers, there are restrictions that vary by project, described in the following table.

4. **Special Cases for Byte Operations**
   1. When the destination type is byte (UB or B) only a 'raw move' using the *mov* instruction supports a packed byte destination register region: *Dst.HorzStride* = 1 and *Dst.DstType* = (UB or B). This packed byte destination register region is not allowed for any other instructions, including a 'raw move' using the *sel*instruction, because the *sel* instruction is based on Word or DWord wide execution channels.

2. There is a relaxed alignment rule for byte destinations. When the destination type is byte (UB or B), destination data types can be aligned to either the lowest byte or the second lowest byte of the execution channel. For example, if one of the source operands is in word mode (a signed or unsigned word integer), the execution data type will be signed word integer. In this case the destination data bytes can be either all in the even byte locations or all in the odd byte locations.

   This rule has two implications illustrated by this example:

```
// Example:
mov (8) r10.0<2>:b r11.0<8;8,1>:w
mov (8) r10.1<2>:b r11.0<8;8,1>:w

// Dst.HorzStride must be 2 in the above example so that the destination
// subregisters are aligned to the execution data type, which is :w.
// However, the offset may be .0 or .1.
// This special handling applies to byte destinations ONLY.
```

5. **Special Cases for Word Operations**

   There are some special cases for word operations for specific projects, described in the following table.

   There is a relaxed alignment rule for word destinations. When the destination type is word (UW, W, HF), destination data types can be aligned to either the lowest word or the second lowest word of the execution channel. This means the destination data words can be either all in the even word locations or all in the odd word locations.

```
// Example:
add (8)  r10.0<2>:hf r11.0<8;8,1>:f r12.0<8;8,1>:hf
add (8)  r10.1<2>:hf r11.0<8;8,1>:f r12.0<8;8,1>:hf

// Note: The destination offset may be .0 or .1 although the destination subregister
// is required to be aligned to execution datatype.
```

6. **Special Requirements for Handling Double and Qword Precision Data Types**

   There are special requirements for handling double precision data types that vary by project, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

   All regioning parameters like stride, execution size, and width are in units of element size.

```
// Example:
mov (4) r10.0<1>:df r11.0<4;4,1>:df
// The above instruction moves four double floats.
```

   In case where source or destination datatype is 64b or operation is integer DWord multiply:

1. Register Regioning patterns where register data bit location of the LSB of the channels are changed between source and destination are not supported on Src0 and Src1 except for broadcast of a scalar.

2. Explicit ARF registers except null and accumulator must not be used.

7. **Special Requirements for Handling Mixed Mode Float Operations**

   There are some special requirements for handling mixed mode float operations for specific projects, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

   When source is float or half float from accumulator register and destination is half float with a stride of 1, the source must register aligned. i.e., source must have offset zero.

   No swizzle is allowed when an accumulator is used as an implicit source or an explicit source in an instruction. i.e. when destination is half float with an implicit accumulator source, destination stride needs to be 2.

```
mac(8) r3<2>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf
mov(8) r3<1>:f acc0.0<8;4,2>:hf
```

   Math operations for mixed mode:

   - In Align1, f16 inputs need to be strided

```
math(8) r3<1>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf
```

   Instructions with pure bfloat16 operands are not supported.

   Mixed mode instruction allows bfloat16 operands in the following cases:

- dst, src0 and src1 for 2-source instructions format not involving multiplier (mov, add, cmp, sel).
- dst and src0 for 2-source instructions format involving multiplier (mul, mac etc).
- dst, src0 and src1 for 3-source instructions format (mad).
- Broadcast of bfloat16 scalar is not supported.

   Unpacked bfloat16 destination with stride 2 when register offset is 0 or 1.

   Mixed mode instruction allows bfloat16 operands in the following cases:

- Execution size must not be greater than 8.

   Packed bfloat16 source and destination when register offset is 0 or 8.

8. **Regioning Rules for Register Indirect Addressing**

   Regioning rules for register indirect addressing vary for specific projects, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

- When the execution size and destination regioning parameters require two adjacent registers, these registers are accessed using one index register ONLY.

```
// Example:
mov (16) r[a0.0]:f r10:f
// The above instruction behaves the same as the following two instructions:
mov (8) r[a0.0]:f r10:f
```

```
mov (8) r[a0.0, 8*4]:f r11:f
```

- When the destination requires two registers and the sources are 1x1 indirect mode, the sources must be assembled from two GRF registers accessed by a single index register. The data for each destination GRF register is entirely derived from one source register. This is ensured by appropriate use of regioning parameters. The exception to this is the use of indirect scalar sources, where the same element is used across the execution size.

```
// Example:
// Case (a)
add (16) r[a0.0]:f r[a0.2]:f r[a0.4]:f
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]:f r[a0.4]:f
add (8) r[a0.0, 8*4]:f r[a0.2, 8*4]:f r[a0.4, 8*4]:f
// Note that the immediate for the second instruction is based on regioning.
// In this case, it is 8 DWs.

// Case (b)
add (16) r[a0.0]:ud r[a0.2]<4;8,1>:w r10<8;8,1>:ud
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]<4;8,1>:w r10<8;8,1>:ud
add (8) r[a0.0, 8*4]:f r[a0.2, 4*2]<4;8,1>:w r11<8;8,1>:ud
// Note that the immediate for the second instruction is based on regioning.
// VertStride of 4 with data type of word.

// Case (c):
add (16) r[a0.0]:f r[a0.2]:f r[a0.4]<0;1,0>:f
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]:f r[a0.4]<0;1,0>:f
add (8) r[a0.0, 8*4]:f r[a0.2, 8*4]:f r[a0.4]<0;1,0>:f
// Note that the src1 indirect address does not change.
```

- Indirect addressing on src1 must be a 1x1 indexed region mode.
- When a Vx1 or a VxH addressing mode is used on src0, the destination may use one or two registers.

```
// Example:
// Case (a)
add (16) r[a0.0]<1>:d r[a0.0]<4,1>:ud r16.0<8;8,1>:ud
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]<1>:d r[a0.0]<4,1>:ud r16.0<8;8,1>:ud
add (8) r[a0.0, 8*4]<1>:d r[a0.2]<4,1>:ud r17.0<8;8,1>:ud
// Since the pointer (index register) is incremented every 4 elements
// (width), the second instruction moves from a0.0 to a0.2.

// Case (b)
add (16)  r10.0<2>:uw  r[a0.0, 0]<1,0>:uw  r16.0<8;8,1>:uw
// The above instruction behaves the same as the following two instructions:
add (8)  r10.0<2>:uw  r[a0.0, 0]<1,0>:uw  r16.0<8;8,1>:uw
add (8)  r11.0<2>:uw  r[a0.8, 0]<1,0>:uw  r17.0<8;8,1>:uw
// Since the pointer (index register) is incremented every 1 element
// (width), the second instruction moves from a0.0 to a0.8.
```

Indirect addressing on the destination must be a 1x1 indexed region mode. The destination cannot span more than 2 adjacent GRF registers irrespective of predicates.

Execution size of 32 is NOT supported in Vx1 or VxH modes.

Accumulator as destination is NOT supported in Vx1 or VxH modes.

Vx1 and VxH indirect addressing for BFloat16, Bfloat8 and TFloat32 data must not be used.

9. **Special Restrictions**

   There are some special restrictions on register region access for specific projects, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

Restriction

Conversion between Integer and HF (Half Float) must be DWord-aligned and strided by a DWord on the destination.
`// Example:`
`add (8) r10.0<2>:hf r11.0<8;8,1>:w r12.0<8;8,1>:w`
`// Destination stride must be 2.`
`mov (8) r10.0<2>:w r11.0<8;8,1>:hf`
`// Destination stride must be 2.`

The src, dst overlapping behavior with the second half src and the first half destination to the same register must not be used with any compressed instruction.
In an atomic chain of instructions in a macro the following applies: 1. Only first instruction can have SWSB check dependency. 2. Internal producer-consumer relationship must not be present.
Regioning Rules for Align1 Ternary Operations Width is an implied regioning parameter. 1. Width is 1 when Vertical and Horizontal Strides are both zero (broadcast access). 2. Width is equal to Vertical Stride when Horizontal Stride is zero. 3. Width is equal to Vertical Stride/Horizontal Stride when both Strides are non-zero. 4. Vertical Stride must not be zero if Horizontal Stride is non-zero. This implies Vertical Stride is always greater than Horizontal Stride. 5. For Source 2, if Horizontal Stride is non-zero, then Width is the a register's width of elements (e.g. 8 for a 32-bit data type). Otherwise, if Horizontal Stride is 0, then so is the Vertical (and rule 1 applies). This means Vertical Stride is always 'Width' * 'Horizontal Stride'.
Byte data type is not supported for src1 register regioning. This includes byte broadcast as well.
Byte data type is not supported for src1 register regioning.
In case of all floating-point data types used in destination: 1. Register Regioning patterns where register data bit location of the LSB of the channels are changed between source and destination are not supported on Src0 and Src1 except for broadcast of a scalar. 2. Explicit ARF registers except null and accumulator must not be used.
In case of conversion/math instruction with fp32 execution datatype and fp16 datatype source or destination with stride of 1 (packed) the following apply • fp16 operand must be a GRF register. • Oword boundary must not be crossed by packed fp16 operand, also

Restriction

• Subregister byte offset within the oword must be half of the execution subregister byte offset
`mov (8) r3.0<1>:hf r4.0<1;1,0>:f // conversion`
`mov (4) r3.2<1>:hf r4.2<1;1,0>:f // conversion (2)*sizeof(hf) mod oword = half of 2*sizeof(f)`
`mov (4) r3.11<1>:hf r4.3<1;1,0>:f // conversion (11)*sizeof(hf) mod oword = half of 3*sizeof(f)`
`math(4) r3.3<1>:hf r4.11<1;1,0>:f // conversion (3)*sizeof(hf) mod oword = half of 3*sizeof(f) = (11)*sizeof(hf) mod oword`
`math(8) r3.11<1>:hf r4.3<1;1,0>:f // conversion (11)*sizeof(hf) mod oword = half of 3*sizeof(f) = (3)*sizeof(hf) mod oword`

Src2 Restrictions 1. Byte data type is not supported. 2. Register Regioning patterns where register data bit location of the LSB of the channels are changed between source and destination are not supported except for broadcast of a scalar. 3. The Source 2 operand must be 64-bit aligned except for broadcast of a scalar.
In an atomic instruction sequence forming a macro the following must apply: • Only first instruction can have SWSB check dependency. • Internal producer-consumer relationship must not be present.

Destination Operand Description

Destination Region Parameters

Based on the above restrictions, a subset of register region parameters are sufficient to describe the destination operand:

- Destination Register Origin
  - Destination Register Number and Destination Subregister Number for direct register addressing mode
  - A Scalar Destination Register Index for register-indirect-register addressing mode
- Destination Register 'Region' - Note that destination register region does not have full region description parameters
  - Destination Horizontal Stride

SIMD Execution Control

This section of the BSpec discusses SIMD execution, both with and without predication. See the subtopics for more details.

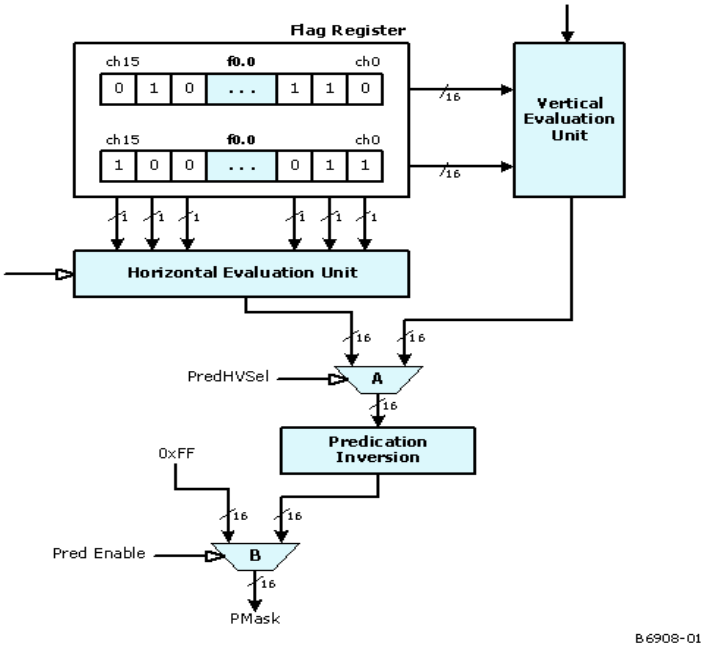
Predication

Predication is the conditional SIMD channel selection for execution on a per instruction basis. It is an efficient way of dynamic SIMD channel enabling without paying branch instruction overhead. When predication is enabled for an instruction, a Predicate Mask (PMask), which contains 16-bit channel enables, is generated internally in EU. Note that PMask is not a software visible register. It is provided here to explain how SIMD execution control works. PMask generation is based on the Predication Control (*PredCtrl*) field, Predication Inversion (*PredInv*) field and the flag source register in the instruction word. See Instruction Summary chapter for definition of these fields.

Predication shows the block diagram of the hardware logic to generate PMask. PMask is generated based on combinatory logic operation of the bits in the flag register. Instruction field *PredCtrl* controls the horizontal evaluation unit and vertical evaluation unit. MUX A in the figure selects whether horizontally evaluated results or vertically-evaluated results are sent to the Predication Invertion unit. The *PredInv* field controls the Prediction Inversion unit. Either one 16-bit flag subregister or the whole flag register may be selected to generate the PMask depending on the predication control modes. MUX B indicates that predication can be enabled and disabled. Predication can be grouped into the following three categories. Predication functionality also depends on the Access Mode of the instruction.

- No predication: Of course, predication can be disabled. This is the most commonly used case.
- Predication with horizontal combination: the predicate mask is generated based on combinatory logic operation of bits within a selected flag subregister.
- Predication with vertical combination: the predicate mask is generated based on combinatory logic operation of bits across flag multiple subregisters.

Generation of predication mask



No Predication

When PredCtrl field of a given instruction is set to 0 ("no predication"), it indicates that no predication is applied to this instruction. Effectively, the resulting PMask is all 1's. This is shown by the 2:1 multiplexer B controlled by the Pred Enable signal in *Predication*. Where predication is not enabled for an instruction, multiplex B is selected to output 0xFF to PMask.

Predication with Horizontal Combination

Predication with horizontal combination inputs the 16 bits of a single flag subregister (f0.0:uw or f0.1:uw) and passes them through combinatory logic of the Horizontal Evaluation unit to create PMask.

The simplest combination is 'no combination' - the same 16 bits from selected flag subregister are output to MUX A. In this case, a bit in the selected flag subregister controls the conditional execution of the corresponding execution channel. Let the selected flag subregister be denoted as f0.#, the following pseudo code describes the predicate mask generation for predication with sequential flag channel mapping.

```
If (PredCtrl == "Sequential flag channel mapping") {
    For (ch=0; ch<16; ch++)
        PMask[ch] = (PredInv == TRUE) ? ~f0.#[ch] : f0.#[ch];
}
```

More complex horizontal evaluation is based on channel grouping. A group of adjacent channels (bits from flag subregister) are evaluated together and a single bit is replicated to the group. The size of groups is in power of 2. The supported combination depends on the Access Mode of an instruction.

In **Align16** access mode, horizontal combination is based on 4-channel groups.

- Channel replication: PredCtrl of '.x', '.y', '.z' and '.w' select a single channel from each 4-channel group and replicate it as the output for the group. For example, PredCtrl = '.x' means that channel 0 in each group is replicated.
- OR combination: PredCtrl of '.any4h' means that if **any** of the channel in a group is enabled, outputs for the 4 channels in the group are all enabled.
- AND combination: PredCtrl of '.all4h' means that only when **all** of the channels in a group are enabled, the output for the group is enabled.

These combinations in **Align16** mode can be described by the following pseudo-code.

```
If (Access Mode == Align16) {
    For (ch = 0; ch < 16; ch += 4)
        Switch (PredCtrl) {
            Case     '.x':   bTmp = f0.#[ch]; break;
            Case     '.y':   bTmp = f0.#[ch+1]; break;
            Case     '.z':   bTmp = f0.#[ch+2]; break;
            Case     '.w':   bTmp = f0.#[ch+3]; break;
            Case '.any4h':   bTmp = f0.#[ch] | f0.#[ch+1] | f0.#[ch+2] | f0.#[ch+3]; break;
Case '.all4h':   bTmp = f0.#[ch] & f0.#[ch+1] & f0.#[ch+2] & f0.#[ch+3]; break;
        }
        bTmp = (PredInv == TRUE) ? ~bTmp : bTmp;
        PMask[ch] = PMask[ch+1] = PMask[ch+2] = PMask[ch+3] = bTmp;
    }
}
```

In **Align1** access mode, horizontal combination is based on AND combination '.any#h' and OR combination '.all#h' on channel groups with various sizes, where # is the number of channels in a group ranging from 2 to 16. This is described by the following pseudo-code.

```
If (Access Mode == Align1) {
    Switch (PredCtrl) {
        Case  '.any2h':   groupSize = 2; <op> = '|'; break;         Case  '.all2h':
groupSize = 2; <op> = '&'; break;
        Case  '.any4h':   groupSize = 4; <op> = '|'; break;         Case  '.all4h':
groupSize = 4; <op> = '&'; break;
        Case  '.any8h':   groupSize = 8; <op> = '|'; break;         Case  '.all8h':
groupSize = 8; <op> = '&'; break;
        Case '.any16h':   groupSize = 16; <op> = '|'; break;         Case '.all16h':
groupSize = 16; <op> = ' break;     }
    For (ch = 0; ch < 16; ch += groupSize) {
        For (inc = 0, bTmp = FALSE; inc < groupSize; inc ++)
            bTmp = bTmp <op> f0.#[ch+inc];
        For (inc = 0; inc < groupSize; inc ++)
            PMask[ch+inc] = bTmp;
    }
}
```

Predication with Vertical Combination

Predication with vertical combination uses both flag subregister as inputs. The AND or OR combination is across the subregisters on a channel by channel basis. This is shown by the following pseudo-code.

```
If (Access Mode == Align1) {
    For (ch = 0; ch < 16; ch ++) {
        If (PredCtrl == 'any2v')
            PMask[ch] = f0.0[ch] | f0.1[ch]
        Else If (PredCtrl == 'all2v')
            PMask[ch] = f0.0[ch] & f0.1[ch]
    }
}
```

Predication with Vertical Combination

Predication with vertical combination uses both flag register as inputs. The AND or OR combination is across the registers on a channel by channel basis. This is shown by the following pseudo-code.

```
If (Access Mode == Align1) {
    For (ch = 0; ch < 32; ch ++) {
        If (PredCtrl == 'anyv')
            PMask[ch] = f0.0[ch] | f1.0[ch]
        Else If (PredCtrl == 'allv')
            PMask[ch] = f0.0[ch] & f1.0[ch]
    }
}
```

End of Thread

There is no special instruction opcode (such as an END instruction) to cause the thread to terminate execution. Instead, the end of thread is signified by a *send* instruction with the end-of-thread (EOT) sideband bit set. Upon executing a *send* instruction with EOT set, the EU stops on the thread. Upon observing an EOT signal on the output message bus, the Thread Dispatcher makes the thread's resource available. If a thread uses pre-allocated resource managed by a fixed function, such as URB handles and

scratch memory, some fixed function protocol also requires the thread to terminate with the message header phase to carry the information in order for the fixed function to release the pre-allocated resource.

EU hardware guarantees that if a terminated thread has in-flight read messages or loads at the time of 'end' that their writebacks will not interfere with either other threads in the system or new threads loaded in the system in the future.

More details can be found in the *send* instruction description in Instruction Reference chapter.

Assigning Conditional Flags

Instructions can output two sets of conditional signals, one set from the computed result before the outputs clamping/re-normalizing/format conversion logic, we call this the pre conditional signals. The second set is generated from the final results after clamping and re-normalizing/format conversion logic, and we call this the post conditional signals. The post conditional signals are used for fusing the DirectX compare instruction. **Note:** In IEEE Mode, the flags generated from the post conditional signals should be equivalent to the flags generated by a separate *cmp* instruction after the current arithmetic instruction; but in ALT mode, this may not be true especially when the computed result is NAN and ALT mode would clamp the NAN to Zero for final results, but NC bit are set to be 1 in the 2nd table below.

The pre conditional signals are used to generated flags for *cmp*/*cmpn* instructions only, this logically does the compare of the two input sources. The post conditional signals are used to generated flags for all the other arithmetic instructions, this logically does the compare of the result with zero.

cmpn with both sources as NaNs is a don't care case as this doesn't impact the MIN/MAX operations.

The pre conditional signals include the following:

- **pre_sign** bit: This bit reflects the sign of the computed result before going through any kind of clamping, normalizing, or format conversion logic.
- **pre_zero** bit: This bit reflects whether the computed result is zero before any kind of clamping, normalizing, or format conversion logic.

The post conditional signals include the following:

- **post_sign** bit: This bit reflects the sign of the final result after all the clamping, normalizing, or format conversion logic.
- **post_zero** bit: This bit reflects whether the final result is zero after all the clamping, normalizing, or format conversion logic. Note: in ALT mode computed NAN result get clamped to ZERO for final result
- **OF** bit: This bit reflects whether an overflow occured in any of the computation of the current instruction, including clamping, re-normalizing, and format conversion.
- **NC** bit: The NaN computed bit indicates whether the computed result is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0. Note: in ALT mode, if the computed result is NAN, NC bit set to 1, but the finial result get clamped to ZERO.

- **NS0** bit: The NaN Source 0 bit indicates whether src0 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0.
- **NS1** bit: The NaN Source 1 bit indicates whether src1 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always 0. For an operation with one source operand, this bit is also set to 0. This bit is only used for the comparison instruction *cmpn*, which is specifically provided to emulate MIN/MAX operations. For any other instructions, this bit is undefined.

Note: the following definition is compliant with DX11 spec especially regarding NAN behavior:

- The comparisons EQ, GT, GE, LT, and LE, when either or both operands is NaN returns FALSE.
- The comparison NE, when either or both operands is NaN returns TRUE.

Flag Generation for *cmp* Instructions (The Supported Conditional Modifiers are .e, .ne, .g, .ge, .l, and .le.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
.e	Equal-to	**(pre_zero & ! (NS0 | NS1)).** This conditional modifier tests whether the two sources are equal. If either source is NaN (i.e. NC is true), the flag is forced to false.
.ne	Not-Equal-to	**! (pre_zero & ! (NS0 | NS1)).** This conditional modifier test whether the two sources are equal. It takes exactly the reverse polarity as the modifier **.e**.
.g	Greater-than	**(! pre_sign & ! pre_zero & ! (NS0 | NS1)).** This conditional modifier tests whether src0 is greater than src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.ge	Greater-than-or-equal-to	**((! pre_sign | pre_zero) & ! (NS0 | NS1)).** This conditional modifier tests whether src0 is greater than or equal to src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.l	Less-than	**(pre_sign & !pre_zero & ! (NS0 | NS1)).** This conditional modifier tests whether src0 is less than src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.le	Less-than-or-equal-to	**((pre_sign | pre_zero) & ! (NS0 | NS1)).** This conditional modifier tests whether src0 is less than or equal to src1. If either source is a NaN (i.e. NC is true), the flag is forced to false.

Flag Generation for All Instructions Other than *cmp/cmpn* Instructions (The Supported Conditional Modifiers are .e, .ne, .g, .ge, .l, .le, .o, and .u.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
.e	Equal-to	**(post_zero & ! NC).** This conditional modifier tests whether the result is equal to zero. If either source is NaN (i.e. NC is true), the flag is forced to false.
.ne	Not-Equal-to	**! (post_zero & ! NC).** This conditional modifier test whether the result is not equal to zero. It takes exactly the reverse polarity as modifier **.e**.
.g	Greater-than	**(! post_sign & ! post_zero & ! NC).** This conditional modifier tests whether result is greater than zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.ge	Greater-than-or-equal-to	**((! post_sign | post_zero) & ! NC).** This conditional modifier tests whether result is greater than or equal to zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.l	Less-than	**(post_sign & ! post_zero & ! NC).** This conditional modifier tests whether result is equal to zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.le	Less-than-or-equal-to	**((post_sign | post_zero) & ! NC).** This conditional modifier tests whether result is equal to or less than zero. If either source is a NaN (i.e. NC is true), the flag is forced to false.
.o	Overflow	**OF.** This conditional modifier tests whether the computed result causes overflow - the computed result is outside the range of the destination data type. Note: the computed result here should represent the infinite precision for intermediate result before converted to destination result for both float and integer operation. Note: The legacy condition modifier behavior is different from IEEE exception Overflow flag. For inf float to int conversion, .o will set the legacy Overflow flag, but IEEE exception Overflow flag won't be set. **Programming Note** **Errata:** Move with double float to integer may not set this flag correctly. All other internal conditional signals are ignored.

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
.u	Unordered	**NC.** This conditional modifier tests whether the computed result is a NaN (unordered). All other internal conditional signals are ignored.

Destination Hazard

Architecture has built-in hardware to avoid destination hazard.

Destination Hazard stands for the risk condition when multiple operations are trying to write to the same destination and the result of the destination may be ambiguous. This may or may not happen on GFX for two instructions with the same destination, or with destinations that have overlapped register region, depending on the ordering of the arrival of destination results. Let's consider two instructions in a thread with potential destination hazard. There may be other instruction between them as long as there is no instruction sourcing the same destination. Using register scoreboards, hardware automatically takes care of the destination hazard by not issuing the second instruction until the destination scoreboard is cleared. However, for certain cases, in fact for most cases, such destination hazard indicated by the register scoreboard is false, causing unnecessary delay of instruction issuing. This may result in lower performance. The destination dependency control field in the instruction word {*NoDDClr, NoDDChk*} allows software to selectively override such hardware destination dependency mechanism. Such performance optimization hooks must be used with extreme caution. When it is not certain that it is a false destination hazard, the programmer should rely on hardware to resolve the dependency.

As the destination dependency control field does not apply to *send* instruction, there is only one condition that a programmer may use the {*NoDDClr, NoDDChk*} capability.

Instructions other than send, may use this control as long as operations that have different pipeline latencies are not mixed. The operations that have longer latencies are:

- Opcodes *pln*, *lrp*, *dp**.
- Operations involving double precision computation.

Integer DW multiplication where both source operands are DWs.

Non-present Operands

Some instructions do not have two source operands and one destination operand. If an operand is not present for an instruction the operand field in the binary instruction must be filed with null. Otherwise, results are unpredictable.

Specifically, for instructions with a single source, it only uses the first source operand src0. In this case, the second source operand src1 must be set to null and also with the same type as the first source operand src0. It is a special case when src0 is an immediate, as an immediate src0 uses DW3 of the instruction word, which is normally used by src1. In this case, src1 must be programmed with register file ARF and the same data type as src0.

Instruction Prefetch

Due to prefetch of the instruction stream, the EUs may attempt to access up to 8 cachelines (512b) beyond the end of the kernel program - possibly into the next memory page. Although these instructions will not be executed, they must be accounted for the prefetch in order to avoid invalid page access faults. GFX software is required to pad the end of all kernel programs with 512b data. A more efficient approach would be to ensure that the page after all kernel programs is at least valid (even if mapped to a dummy page). Note that the **General State Access Upper Bound** field of the STATE_BASE_ADDRESS command can be used to prevent memory accesses past the end of the General State heap (where kernel programs must reside).

ISA Introduction

ISA Introduction

This chapter contains these sections that introduce this volume.

- Introducing the Execution Unit
- EU Terms and Acronyms
- EU Changes by Processor Generation
- EU Notation

Subsequent chapters cover:

- EU Data Types
- Execution Environment
- Exceptions
- Instruction Set Summary
- Instruction Set Reference
- EU Programming Guide

The EU Programming Guide provides some useful examples and information but is not a complete or comprehensive programming guide.

Introducing the Execution Unit

This section introduces the Execution Unit (EU), a simple and capable processor within the GPU that supports graphics processing within the graphics pipelines, can do general purpose computing (GPGPU), and responds to exceptional conditions via the System Routine.

The EU provides parallelism at two levels: thread and data element. Multiple threads can execute on the EU; the number executing concurrently depends on the processor and is transparent to EU code. Each thread has its own registers (GRF and ARF, described below). Most EU instructions operate on arrays of data elements; the number of data elements is normally the *ExecSize (execution size)* or number of channels for the instruction. A *channel* is a logical unit of execution for data element access, masking,

and flow control within instructions. The number of channels is independent of the number of physical ALUs or FPUs for a particular graphics processor.

EU native instructions are 128 bits (16 bytes) wide. Some combinations of instruction options can use compact instruction formats that are 64 bits (8 bytes) wide. Identifying instructions that can be compacted and creating the compact representations is done by software tools, including compilers and assemblers.

Data manipulation instructions have a destination operand (dst) and one, two, or three source operands (src0, src1, or src2). The instruction opcode determines the number of source operands. An instruction's last source operand can be an immediate value rather than a register.

Data read or written by a thread is generally in the thread's *GRF (General Register File)*, 128 general registers, each 32 bytes. A data element address within the GRF is denoted by a register number (r0 to r127) and a subregister number. In the instruction syntax, subregister numbers are in units of data element size. For example, a :d (Signed Doubleword Integer) element can be in subregister 0 to 7, corresponding to byte numbers in the instruction encoding of 0, 4, ... 28.

The EU cannot directly read or write data in system memory.

Specialized registers used to implement the ISA are in a distinct per thread *Architecture Register File (ARF)*. Each such register or group of related registers has its own distinct name. For example, ip is the instruction pointer and f0 is a flags register. An ARF register can be a src0 or dst operand but not a src1 or src2 operand. There are restrictions on how particular ARF registers are accessed that should be understood before directly reading or writing those registers. See the ARF Registers section for more information.

The EU supports both integer and floating-point data types, as described in the Numeric Data Types section.

For EU flow control, each channel has its own per-channel instruction pointer (PcIP[n]) and only executes an instruction when IP == PcIP[n] and any other masks enable the channel. Most flow control instructions use signed offsets from the current instruction address to reference their targets. Unconditional branches are done using *mov* with IP as the destination. Flow control can also use SPF (Single Program Flow) mode to execute with a single instruction pointer (IP).

The EU ISA supports predication, masking, regioning, swizzling, some type conversions, source modification, saturation, accumulator updates, and flag updates as part of instruction execution:

- *Predication* creates a bit mask (*PMask*) to enable or disable channels for a particular instruction execution. Pmask is derived from flag register and subregister values using boolean formulas determined by the PredCtrl (Predicate Control) and PredInv (Predicate Inversion) instruction fields. See the Predication section.
- *Masking* is the overall process of determining which channels execute for a given instruction based on five factors:
    - Number of channels (only channels in [0, *ExecSize* - 1] can execute)
    - Execution mask (*EMask*)

- Whether the channel is on the instruction (if not in Single Program Flow mode and MaskCtrl is not NoMask)
- Predicate mask (*PMask*)
- In Align16 mode, any enabling of channels using the Dst.ChanEn instruction field (if MaskCtrl is not NoMask).

- *Regioning* specifies an array of data elements contained in one or two registers, with options for scattering, interleaving, or repeating data elements in registers using width and stride values, subject to significant constraints. Regioning also includes access mode (Align1 or Align16) and addressing mode (Direct or Indirect). See the Registers and Register Regions section.
- *Swizzling* allows small scale reordering of data elements within groups of four at the input using the modulo 4 channel names x, y, z, and w. For example, a swizzle of .wzyx with an *ExecSize* of 8 reads execution channels 0 to 7 from these input channels: 3, 2, 1, 0, 7, 6, 5, and 4. Swizzling is only available in the Align16 access mode, described in the Execution Environment chapter.
- *Type Conversions* do any needed conversion from source data type to execution data type and from execution data type to destination data type. See Execution Data Type for more information. Each instruction description indicates what combinations of data types are supported.
- *Source Modification* modifies a source operand just before doing the requested operation. For a numeric operation, the choices are:
  - No modification (normal).
  - - indicating negation.
  - (abs) indicating absolute value.
  - -(abs) indicating a forced negative value.

  Source modification logically occurs after any conversion from source data type to execution data type. Each instruction description indicates whether it supports source modification.
- *Saturation* clamps result values to the nearest value within a saturation range determined by the destination type. For a floating-point type, the saturation range is [0.0, 1.0]. For an integer type, the saturation range is the entire range for that type, for example [0, 65535] for the UW (Unsigned Word) type. Each instruction description indicates whether it supports saturation.
- *Accumulator Updates* optionally update the accumulator register or registers in the ARF with destination values as a side effect of instruction execution. The AccWrCtrl instruction field enables accumulator updates. The Accumulator Disable flag in control register 0 (cr0) can be used to disable accumulator updates, regardless of AccWrCtrl values; for example, this flag may be used in the System Routine.
- *Flag Updates* optionally update a flags register and subregister (f0.0, f0.1, f1.0, or f1.1) with conditional flags based on the CondModifier (Condition Modifier) instruction field. For example, a CondModifier of **.nz** (not zero) assigns flag bits based on whether result elements are not zero (1) or zero (0). Each instruction description indicates whether it supports the Condition Modifier and any restrictions on the values supported.

The EU is not required to execute steps in its internal pipeline sequentially or in order, so long as it produces correct results.

The assembler syntax uses spaces between operands and encloses ExecSize and any predicate in parentheses. Instruction mnemonics, register names, conditional modifiers, predicate controls, and type designators use lowercase. Function names used with the math instruction are UPPERCASE.

```
( pred ) inst cmod sat ( exec_size ) dst src0 src1 { inst_opt, ... }
```

General register destination regions use the syntax r*m.n*<*HorzStride*>:type. General register directly addressed source regions use the syntax r*m.n*<*VertStride*;*Width*,*HorzStride*>:type. You need to understand more about register regioning to understand all of these terms.

The following example assembly language instruction adds two packed 16-element single-precision Float arrays in r4/r5 and r2/r3 writing results to r0/r1, only on those channels enabled by the predicate in f0.0 along with any other applicable masks.

```
(f0.0) add (16) r0.0<1>:f r2.0<8;8,1>:f r4.0<8;8,1>:f
```

EU Terms and Acronyms

This section provides three tables describing EU general terms and acronyms, EU data types, and EU selected ARF registers.

EU General Terms and Acronyms

Term	Description
ALT mode	A floating-point execution mode that maps +/- inf to +/- fmax, +/- denorm to +/-0, and NaN to +0 at the FPU inputs and never produces infinities, denormals, or NaN values as outputs. See IEEE mode.
ALU	Arithmetic Logic Unit. A functional block that performs integer arithmetic and logic operations, as distinct from instruction fetch and decode, floating-point operations (see FPU), or messaging.
AOS	Array Of Structures. Also see SOA.
ARF	Architecture Register File, a distinct register file containing registers used to implement specific ISA features. For example the Instruction Pointer and condition flags are in ARF registers. See GRF.
byte	An 8-bit value aligned on an 8-bit boundary and the basic unit of addressing. Bits within a byte are denoted 0 to 7 from LSB to MSB.
channel	A logical unit of SIMD data parallel execution within a thread and within the EU. The number of physical ALUs or FPUs is not directly related to the number of channels. Up to 32 channels are supported.
compact instruction	A 64-bit instruction encoded as described in the EU Compact Instructions section. Only some combinations of instruction parameters can be encoded as compact instructions. See native instruction.
compressed instruction	An instruction that writes to two destination registers. For example a SIMD16 instruction with Float operands can write channels 0 to 7 to one 32-byte general register and channels 8 to 15 to a second, consecutive 32-byte general register.
denorm	A very small but nonzero number in IEEE mode, with a magnitude less than the smallest normalized floating-point number representable in a particular floating-point format. Denormals lose precision

Term	Description
	as their values approach zero, called *gradual underflow*.
DWord	Doubleword. A 32-bit (4-byte) value aligned on a 32-bit (4-byte) boundary. Bits within a DWord are denoted 0 to 31 from LSB to MSB.
EOT	End of Thread. A flag set on a *send* or *sendc* instruction to terminate a thread's execution on the EU.
EU	Execution Unit. The single GPU unit described in this volume. This volume describes individual data parallel execution paths within a thread in the EU as *channels*. A few fields, like EUID, use EU to refer to a particular hardware resource used to implement the overall EU.
exception	An error or interrupt condition that arises during execution that may transfer control to the System Routine. Some exceptions can be disabled, preventing such transfers. As defined in this volume, some errors do not produce exceptions.
ExecSize	The number of execution channels for a particular instruction. Channels within that number are enabled or disabled by various masks.
floating-point	Numeric types that allow fractional values and often a wider range than integer types. The EU supports binary floating-point types including the single precision type and the double precision type defined by the IEEE 754 standard.
GRF	General Register File, a distinct register file containing 128 general registers, r0 to r127. Each general register is 256 bits (32 bytes), can contain any type of data, and can be accessed with any valid combination of addressing mode, access mode, and region parameters. A general register is directly addressed using a register number and subregister number, or indirectly addressed using an address subregister (index register) and an address immediate offset.
IEEE mode	A floating-point execution mode that supports all the kinds of floating-point values described by the IEEE 754 standard: normalized finite nonzero binary floating-point numbers, signed zeros, signed infinities, signed denormals that are closer to zero than any normalized value but still nonzero, and NaN (not a number) values. See ALT mode.
index register	An address subregister when used for indirect addressing.
inf	Infinity, +inf or -inf, as a floating-point value in IEEE mode.
instruction	In this volume, *instruction* always refers to an EU instruction.
ISA	Instruction Set Architecture, processor aspects visible to programs and programmers and independent of a particular implementation, including data types, registers, memory access, addressing modes, exceptions, instruction encodings, and the instruction set itself. An ISA does not include instruction timing, hardware pipeline details, or the number of physical resources (ALUs, FPUs, instruction decoders) mapped to logical constructs (threads, channels). This volume also includes a recommended assembly language syntax, closely related to the ISA but logically distinct from it.
LSB	Least significant bit.
message	A data structure transmitted from a thread to another thread, to a shared function, or to a fixed function. Message passing is the primary communication mechanism of the GFX architecture.
MSB	Most significant bit.
NaN	Not a Number. A non-numeric value allowed in the standard single precision and double precision floating-point number formats. Quiet NaNs propagate through calculations and signaling NaNs cause exceptions. NaNs are not used in the ALT floating-point mode.
native	A 128-bit instruction, the regular instruction format that allows all defined instruction parameters

Term	Description
instruction	and options. Some instructions can also be encoded using a 64-bit compact instruction format.
OWord	Octword. A 128-bit (16-byte) value aligned on a 128-bit (16-byte) boundary. Bits within an OWord are denoted 0 to 127 from LSB to MSB. This term is used rarely and may be dropped from future versions of this volume.
packed	A register region is described as *packed* if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist. The immediate vector data types are all described as *Packed* because each such type packs several small data elements into a 32-bit immediate value.
QWord	Quadword. A 64-bit (8-byte) value aligned on a 64-bit (8-byte) boundary. Bits within a QWord are denoted 0 to 63 from LSB to MSB.
region	A collection of data locations in registers and subregisters for a source or destination operand. The associated regioning parameters allow regions to be arrays with various layouts.
register	Part of the directly accessible state of an EU program, such as a general register in the GRF or an architecture register in the ARF. Note that system memory is not directly accessible.
SIMD	Single Instruction Multiple Data. Each EU instruction can operate on multiple data elements in parallel, as specified by the instruction's ExecSize.
SIP	System Instruction Pointer, the starting IP value for the System Routine.
SOA	Structure of Arrays. Also see AOS.
SPF	Single Program Flow. A mode in which every execution channel uses the common instruction pointer, IP in the ip register. The SPF bit in the control register is 1 to enable SPF and 0 to disable it. If SPF is disabled, then each execution channel n has its own instruction pointer, PcIP[n] and each channel n is only eligible to execute, subject to other masking, when PcIP[n] == IP.
swizzle	Rearrange data elements within a vector. The EU supports modulo four swizzling of register source operands at the input in the Align16 access mode.
System Routine	A global EU exception handling routine. Any enabled exception from any EU thread transfers control to this routine.
thread	An instance of a program executing on the EU. The life cycle for a thread on the EU starts with the first instruction after being dispatched to the EU by the Thread Dispatcher and ends after executing a *send* or *sendc* instruction with EOT set, signaling thread termination. Threads can be independent or can communicate with each other via the Message Gateway shared function.
word	A 16-bit (2-byte) value aligned on a 16-bit (2-byte) boundary. Bits within a word are denoted 0 to 15 from LSB to MSB. *Word* has denoted a 16-bit unit for Intel processors since the 8086 and 8088 processors were introduced in 1978.

The next table lists all EU numeric data types. See the Numeric Data Types section for more information about each data type.

EU Numeric Data Types (Listed Alphabetically by Short Name)

Short Name	Assembler Syntax	Long Name	Size in Bytes	Size in Bits	Integral or Float	Description
B	:b	Signed Byte Integer	1	8	I	Signed integer in the range -128 to 127.
D	:d	Signed Doubleword Integer	4	32	I	Signed integer in the range -2^{31} to 2^{31} - 1.
DF	:df	Double Float	8	64	F	Double precision floating-point number.
F	:f	Float	4	32	F	Single precision floating-point number.
HF	:hf	Half Float	2	16	F	Half precision floating-point number.
Q	:q	Signed Quadword Integer	8	64	I	Signed integer in the range -2^{63} to 2^{63} - 1.
UB	:ub	Unsigned Byte Integer	1	8	I	Unsigned integer in the range 0 to 255.
UD	:ud	Unsigned Doubleword Integer	4	32	I	Unsigned integer in the range 0 to 2^{32} - 1.
UQ	:uq	Unsigned Quadword Integer	8	64	I	Unsigned integer in the range 0 to 2^{64} - 1.
UV	:uv	Packed Unsigned Half Byte Integer Vector	4	32	I	Eight 4-bit unsigned integer values each in the range 0 to 15. Only used as an immediate value.
UW	:uw	Unsigned Word Integer	2	16	I	Unsigned integer in the range 0 to 65,535.
V	:v	Packed Signed Half Byte Integer Vector	4	32	I	Eight 4-bit signed integer values each in the range -8 to 7. Only used as an immediate value.
VF	:vf	Packed Restricted Float Vector	4	32	F	Four 8-bit restricted float values. Only used as an immediate value.
W	:w	Signed Word Integer	2	16	I	Signed integer in the range -32,768 to 32,767.

The next table lists the seven ARF registers that you should understand first, omitting several others. See the ARF Registers section for more information, including descriptions of additional registers not listed below.

EU Selected ARF Registers (Listed Alphabetically by Name)

Name	Assembler Syntax	Description
Accumulators	acc0, acc1	Data registers that can hold integer or floating-point values of various sizes. Many instructions can implicitly update accumulators with a copy of destination values, done by setting the AccWrCtrl instruction option. A few instructions, like *mac* (Multiply Accumulate), use the accumulators as an implicit source operand, useful for some iterative calculations. There are added accumulator registers acc2 to acc9 for special purposes; these added accumulators are not generally used.
Address Register	a0.*s*	Holds subregisters primarily used for indirect addressing. Each subregister is a 16-bit UW (Unsigned Word) value. For an indirectly addressed operand or element, the subregister value plus an AddrImm signed offset field determines the byte address (RegNum and SubRegNum) within the register file (GRF). There are 16 address subregisters.
Control Register	cr0.*s*	Contains bit fields for floating-point modes, flow control modes, and exception enable/disable. Also contains exception indicator flags and saves the AIP (Application Instruction Pointer) on transferring control to the System Routine to handle an exception.
Flags	f*r.s*	Used as the outputs for various channel conditional signals, such as equality/zero or overflow. Used as the inputs for predication. There are two 32-bit flags registers each containing two 16-bit subregisters.
Instruction Pointer (IP)	ip	References the current instruction in memory, as an unsigned offset from the General State Base Address. IP is the thread's overall instruction pointer. Each channel n can have its own instruction pointer (PcIP[n]). If not in Single Program Flow mode (SPF is 0) then only those channels where PcIP[n] == IP are eligible to execute the instruction, if enabled by all other applicable masks.
Null Register	null	Indicates a non-existent operand. Unused operands in the instruction format, like the unused second source operand field in a *mov* instruction, are encoded as null. For present source operands, reading a null source operand returns undefined values. For null destination operands, results are discarded but any implicit updates to accumulators or flags still occur.
State Register	sr0.*s*	Contains thread identification and scheduling fields, and mask fields for enabling or disabling channels.

Execution Units (EUs)

Each EU is a vector machine capable of performing a given operation on as many as 16 pieces of data of the same type in parallel (though not necessarily on the same instant in time). In addition, each EU can support a number of execution contexts called *threads* that are used to avoid stalling the EU during a high-latency operation (external to the EU) by providing an opportunity for the EU to switch to a completely different workload with minimal latency while waiting for the high-latency operation to complete.

For example, if a program executing on an EU requires a texture read by the sampling engine, the EU may not necessarily idle while the data is fetched from memory, arranged, filtered and returned to the EU. Instead, the EU will likely switch execution to another (unrelated) thread associated with that EU. If that thread encounters a stall, the EU may switch to yet another thread and so on.

Once the Sampler result arrives back at the EU, the EU can switch back to the original thread and use the returned data as it continues execution of that thread.

The fact that there are multiple EU cores each with multiple threads can generally be ignored by software. There are some exceptions to this rule: e.g., for:

Description
thread-to-thread communication (see *Message Gateway*, *Media*)
synchronization of thread output to memory buffers (see *Geometry Shader*)

In contrast, the internal SIMD aspects of the EU are very much exposed to software.

This volume will not deal with the details of the EUs.

EU Changes by Processor Generation

This section describes how the EU changes for particular processor generations. Instruction compaction tables can differ for each generation, so that is not mentioned in these lists. Particular readers and audiences can see only certain content in this section. Workarounds for particular generations, SKUs, or steppings are not included in these lists. Some small changes in instruction layouts are not included in these lists.

Description
These features or behaviors have been added: • The maximum *ExecSize* increases to 32, for byte or word operands. • Increase the number of flag registers from one to two. • Add the NibCtrl field, used with QtrCtrl to select groups of channels or flags. • Add the DF (Double Float) data type, the first time an 8-byte data type is supported. DF only supports the IEEE floating-point mode and not the ALT floating-point mode. • Add a shared source data type field and a destination data type field for instructions with three source operands, allowing F (Float), DF (Double Float), D (Signed Doubleword Integer), or UD (Unsigned Doubleword Integer) types to be specified.

Description
• Add bit manipulation instructions: *bfi1*, *bfi2*, *bfrev*, *cbit*, *fbh*, and *fbl*. • Add the integer *addc* (Add with Carry) and *subb* (Subtract with Borrow) instructions. • Add the *brc* (Branch Converging) and *brd* (Branch Diverging) instructions. • For the *cmp* and *cmpn* instructions, relax the accumulator restrictions. • For the *sel* instruction, remove the accumulator restriction. • Add the Rounding Mode and Double Precision Denorm Mode fields in Control Register 0.
These features or behaviors have been added: • DF (Double Float) operands use an element size of 8. Regioning and channel parameters for the DF type are determined normally, in the same way as for other types. • Add the channel enable register, flow control registers, and stack pointer register in the ARF. • In the Control Register, add the Force Exception Status and Control, Context Save Status, and Context Restore Status bits. • Relative instruction offsets (JIP, UIP) are now 32-bit values in units of bytes (rather than 16-bit values using 8-byte units) for some instructions: *brc*, *brd*, *call*, and *jmpi*. • A *call* instruction can get the relative instruction offset (JIP) from a register. • Add the *calla* (Call Absolute) instruction. • A *mov* instruction with different source and destination types can now use conditional modifiers.
These features or behaviors have been added: • Add the HF (Half Float) type and a corresponding HF execution data type and execution path. • Add flags to indicate IEEE floating-point exceptions and to enable or disable exception reporting to those flags. • Add the Single Precision Denorm Mode bit in Control Register 0. It can be enabled to allow calculations using the F (Float) type in IEEE floating-point mode to support denormals and gradual underflow. • Add the Q (Signed Quadword Integer) and UQ (Unsigned Quadword Integer) types. Integer source types cannot mix 64-bit and 8-bit operands. Some integer instructions (e.g., *avg*) do not support Q or UQ source types. • Instructions with one source operand and a 64-bit source type can have immediate 64-bit source operands. • The JIP and UIP relative instruction offset fields in all remaining flow control instructions are 32-bit values in units of bytes (rather than 16-bit values using 8-byte units). • The instruction layout is noticeably different. The SrcType and DstType instruction fields are widened to allow for more type encodings as three types are added. The AddrSubRegNum instruction field is widened to allow for 16 address subregisters rather than 8. The layout now supports 64-bit immediate source operands for one-source instructions and 32-bit relative instruction offset fields for flow control instructions. • In the 3-source instruction format, widen the SrcType and DstType fields and add an encoding for the HF (Half Float) type. • Add a compact instruction format for 3-source instructions. • Use a different source modifier interpretation for logical (*and*, *not*, *or*, *xor*) instructions. • An accumulator source operand for a logical instruction can now have a source modifier.

Description
• Add eight address subregisters, increasing the number of address subregisters from 8 to 16. • For the *brc* and *brd* instructions do not allow the **Switch** instruction option. • For the *cmp* and *cmpn* instructions, remove the accumulator restrictions. • Add the *goto* instruction, reusing the opcode for the discontinued *fork* instruction. • Add the *join* instruction. • For the *lzd* instruction, remove the accumulator restriction. • The *mach* instruction reverses the roles of the two source operands compared to previous generations. • Add the *madm* instruction. • Enhance the *math* instruction to allow some immediate source values and support the INVM and RSQRTM functions. • For the *mul* instruction, relax the accumulator restriction on source operands so it applies for only integer source operands. • For the rounding instructions (*rndd*, *rnde*, *rndu*, and *rndz*), remove the accumulator restrictions. • Revise the *shl* and *shr* instructions to use the low 6 bits of the shift count in QWord mode, versus the low 5 bits otherwise. • Add the *smov* instruction. • Add eight accumulator registers, acc2 to acc9, used only for the special purpose of emulating IEEE-compliant fdiv and sqrt operations. • Add message control registers. • Widen the sp (Stack Pointer) register to 64 bits. • Add the IEEE Exception, Page Fault Status, and Page Fault Code bit fields in the State Register. • Remove some regioning restrictions when operands span two registers.
These features or behaviors have been added: • Add the Half Precision Denorm Mode bit in the Control Register. It can be enabled to allow calculations using the HF (Half Float) type to support denormals and gradual underflow.
These features or behaviors have been added: • Remove the Align 16 access mode • Add *dp4a* instruction. • Add *rol* and *ror* instruction for dword datatype. • Remove *pln, line* and *dp* product instruction. • Restriction on source2 regioning - byte datatype support removed, swizzling of data also removed except for broadcast.
These features or behaviors have been added: • Add 8bits of software hints in instruction to facilitate scheduling. • Byte datatype support removed from source1.

Description
These features or behaviors have been added: • Add *bfn* and *add3* instructions. • Add *dpas* and *dpasw* instruction with int8/4/2, bfloat16 and half-float datatypes support. • Increase general accumulators from 2 to 4. • Add support for float32 to bfloat16 datatype conversion. • Mixed mode support removed for half-float and added for bfloat16 datatypes.

EU Notation

The Courier New font is used for code examples and for the Syntax, Format, and Pseudocode sections in the instruction reference.

The *italic* font style is used for instruction mnemonics outside of code (e.g., the *send* instruction), for syntactic production names, for key values in algorithms (*ExecSize*), and to emphasize a word or phrase. For example: When bit 10 is set, the destination register scoreboard is *not* cleared.

The **bold** font weight is used for the short name and long name of a bit field being described, for value names being defined, for syntactic terminals, for unnumbered subheadings, and for the terms Note or Workaround used to introduce a paragraph.

Bit field names and value names used where not being defined and not as syntactic terminals are in plain text.

Bit field values in hex use the 0x prefix. The BSpec currently uses the 0x prefix for hex in some parts and the h suffix for hex in other parts. For single bits, values appear as simply 0 or 1. For multi-bit binary values, the appropriate number of binary digits appears with a b suffix.

Instruction mnemonics are lowercase. Function names invoked using the *math* instruction are UPPERCASE. For example, SQRT.

Device names in the new syntax do not use the Dev prefix or square brackets and often appear in tagged tables with a blue background. For example:

Tables describing bit field layouts or registers proceed from most significant to least significant bits. Figures showing bit fields or registers show most significant bits on the left and least significant bits on the right.

Any bit, field, or register described as Reserved should be regarded as undefined and unpredictable. Such bits should be treated as follows:

- When testing values, do not depend on the state of reserved bits. Mask out or otherwise ignore such bits.
- Sometimes software must initialize reserved bits. For example, a compiler must write complete instruction values when creating an instruction stream, including reserved bits. In such cases, write reserved bits as zeros unless otherwise indicated.
- Do not use reserved bits as extra storage for software-defined values; put nothing in such bits.

- When saving state and restoring state, save and restore any reserved bits as well.
- Do not assume that reserved bits are invariant between explicit writes. Software should function even if reserved bits change in undefined and unpredictable ways.

Any value, encoding, or combination of values or encodings described as Reserved must not be used. The EU's behavior is undefined in this case.

When a combination of instruction parameters or an EU state is described as producing undefined results or behavior, do not assume that undefined results or behavior are confined to specific instructions, operands, registers, or channels.

Execution Environment

EU Data Types

Fundamental Data Types

Numeric Data Types

Floating Point Modes

- IEEE Floating Point Mode
  - Partial Listing of Honored IEEE 754 Rules
  - Complete Listing of Deviations or Additional Requirements vs IEEE 754
  - Min/Max of Floating Point Numbers
- Alternative Floating Point Mode

Floating-Point Support

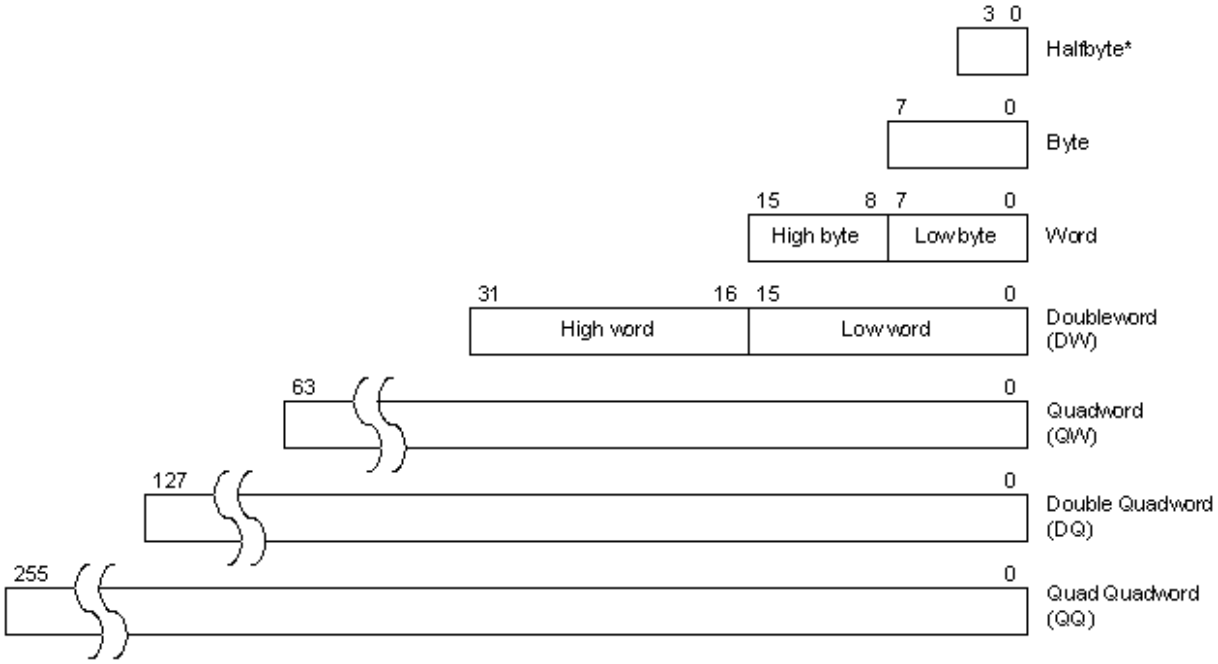
- IEEE Floating-Point Exceptions
- Floating-Point Compare Operations

Type Conversion

Fundamental Data Types

The fundamental data types in the architecture are halfbyte, byte, word, doubleword (DW), quadword (QW), double quadword (DQ) and quad quadword (QQ). They are defined based on the number of bits of the data type, ranging from 4 bits to 256 bits. As shown in the figure below, a halfbyte contains 4 bits, a byte contains 8 bits, a word contains two bytes, a doubleword (DWord) contains two words, and so on. Halfbyte is a special data type that is not accessed directly as a standalone data element; it is only allowed as a subfield of the numeric data type of "packed signed halfbyte integer vector" described in the next section.

Fundamental Data Types



With the exception of halfbyte, the access of a data element to/from a register or to/from memory must be aligned on the natural boundaries of the data type. The natural boundary for a word has an even-numbered address in units of bytes. The natural boundary for a doubleword has an address divisible by 4 bytes. Similarly, the natural boundary for a quadword, double quadword, and quad quadword has an address divisible by 8, 16, and 32 bytes, respectively. Double quadword, and quad quadword do not have corresponding numeric data types. Instead, they are used to describe a group (a vector) of numeric data elements of smaller size aligned to larger natural boundaries.

Numeric Data Types

The numeric data types defined in the architecture include signed and unsigned integers and floating-point numbers (floats) of various sizes. These numeric data types are described below.

Integer Numeric Data Types

The Execution Unit supports the following integer data types. Signed integer types use two's complement representation for negative numbers.

UB: Unsigned Byte, 8-bit Unsigned Integer

7		0

B: Byte, 8-bit Signed Integer

7	6		0

7	6		0
S			

UW: Unsigned Word, 16-bit Unsigned Integer

15		0

W: Word, 16-bit Signed Integer

15	14		0
S			

UD: Unsigned Doubleword, 32-bit Unsigned Integer

31		0

D: Doubleword, 32-bit Signed Integer

31	30		0
S			

UQ: Unsigned Quadword, 64-bit Unsigned Integer

63		0

Q: Quadword, 64-bit Signed Integer

63	62		0
S			

UV: Packed Unsigned Half-Byte Integer Vector, 8 x 4-Bit Unsigned Integer

31		28	27		24	23		20	19		16	15		12	11		8	7		4	3		0

V: Packed Signed Half-Byte Integer Vector, 8 x 4-Bit Signed Integer

31		28	27		24	23		20	19		16	15		12	11		8	7		4	3		0
S			S			S			S			S			S			S			S		

The following table summarizes the EU integer data types.

Execution Unit Integer Data Types

Notation	Size in Bits	Name	Range
UB	8	Unsigned Byte Integer	[0, 255]
B	8	Signed Byte Integer	[-128, 127]
UW	16	Unsigned Word Integer	[0, 65535]
W	16	Signed Word Integer	[-32768, 32767]
UD	32	Unsigned Doubleword Integer	[0, 2^{32} - 1]
D	32	Signed Doubleword Integer	[-2^{31}, 2^{31} - 1]
UQ	64	Unsigned Quadword Integer	[0, 2^{64} - 1]
Q	64	Signed Quadword Integer	[-2^{63}, 2^{63} - 1]
UV	32	Packed Unsigned Half-Byte Integer Vector	[0, 15] in each of eight 4-bit immediate vector elements.
V	32	Packed Signed Half-Byte Integer Vector	[-8, 7] in each of eight 4-bit immediate vector elements.

Restriction: Only a raw move using the *mov* instruction supports a packed byte destination register region. For information about raw moves, refer to the **Description** in *mov - Move*.

Floating-Point Numeric Data Types

The Execution Unit supports the following floating-point data types.

Type or Description
The Float type uses the single precision format specified in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.
The Double Float type uses the double precision format specified in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.
In the ALT floating-point mode, representations for infinities, denorms, and NaNs within those formats are not used. The EU does not support the double extended precision (80-bit) floating-point format found in the x86/x87/Intel 64 floating-point registers. All floating-point formats are signed using signed magnitude representation (a distinct sign bit, separate from the magnitude information).
The Half Float type uses the binary16 format specified in IEEE Standard 754-2008.

Type or Description
The F (Float) type supports both the ALT and IEEE floating-point modes, controlled by the Single Precision Floating-Point Mode bit in the Control Register.
Whether IEEE mode F calculations support denorms or flush denormalized values to zero is controlled by the Single Precision Denorm Mode bit in the Control Register.
The DF (Double Float) type only supports the IEEE floating-point mode. Whether DF calculations support denorms or flush denormalized values to zero is controlled by the Double Precision Denorm Mode bit in the Control Register.
The HF (Half Float) type only supports the IEEE floating-point mode.
Whether HF calculations support denorms or flush denormalized values to zero is controlled by the Half Precision Denorm Mode bit.
BFloat type will be used machine learning computes due to its higher numerical range compared to half float.

HF: Half Float, 16-bit Half-Precision Floating-Point Number

15	14		10	9		0
S	biased exp.			fraction		

BF: BFloat, 16-bit Bfloat Half-Precision Floating-Point Number

15	14		7	6		0
S	biased exp.			fraction		

F: Float, 32-bit Single-Precision Floating-Point Number

31	30		23	22		0
S	biased exponent			fraction		

DF: Double Float, 64-bit Double-Precision Floating-Point Number

63	62		52	51		0
S	biased exponent			fraction		

VF: Packed Restricted Float Vector, 4 x 8-Bit Restricted Precision Floating-Point Number

31	30		28	27		24	23	22		20	19		16	15	14		12	11		8	7	6		4	3		0
S	b. exp.			frac.			S	b. exp.			frac.			S	b. exp.			frac.		S	b. exp.				frac.		

The following table summarizes the EU floating-point data types.

Execution Unit Floating-Point Data Types

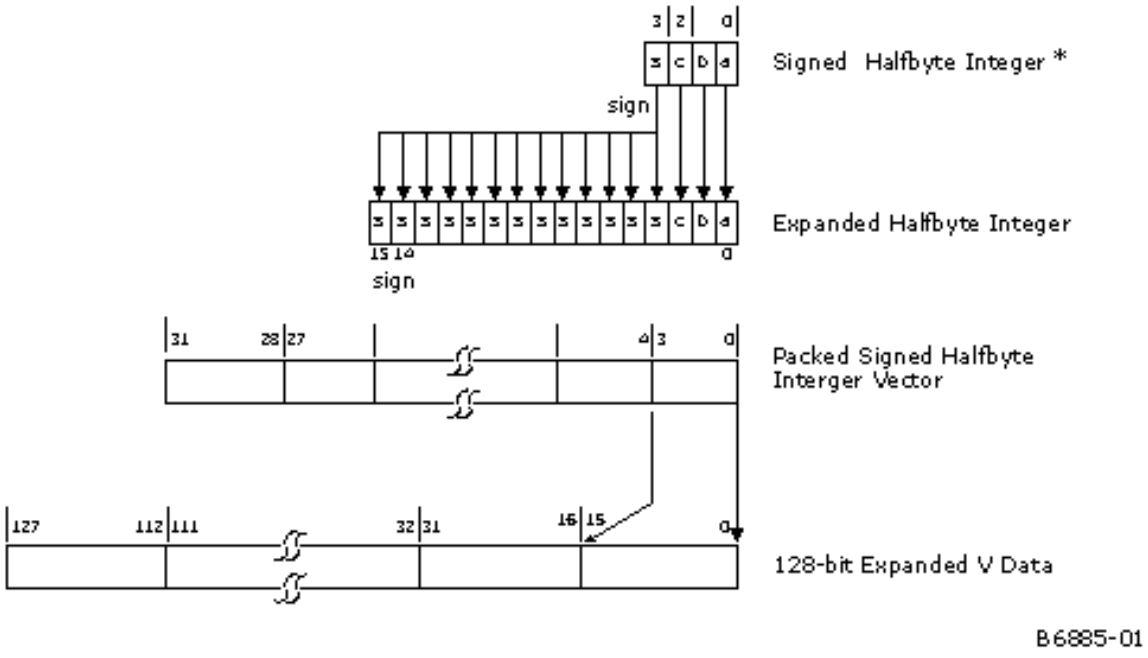
Notation	Size in Bits	Name	Range
HF	16	Half Float	Half precision, 1 sign bit, 5 bits for the biased exponent, and 10 bits for the significand: $[-(2-2^{-10})^{15}...-2^{-24}, 0.0, 2^{-24}... (2-2^{-10})^{15}]$
BF	16	BFloat	Bloaf precision, 1 sign bit, 8 bits for the biased exponent, and 7 bits for the significand: $[-(2-2^{-7})^{127}...-2^{-133}, 0.0, 2^{-133}... (2-2^{-7})^{127}]$
F	32	Float	Single precision, 1 sign bit, 8 bits for the biased exponent, and 23 bits for the significand: $[-(2-2^{-23})^{127}...-2^{-149}, 0.0, 2^{-149}... (2-2^{-23})^{127}]$
DF	64	Double Float	Double precision, 1 sign bit, 11 bits for the biased exponent, and 52 bits for the significand: $[-(2-2^{-52})^{1023}...-2^{-1074}, 0.0, 2^{-1074}... (2-2^{-52})^{1023}]$
VF	32	Packed Restricted Float Vector	Restricted precision. Each of four 8-bit immediate vector elements has 1 sign bit, 3 bits for the biased exponent (bias of 3), and 4 bits for the significand: [-31... -0.1328125, -0, 0, 0.1328125 ... 31]

Packed Signed Half-Byte Integer Vector

A packed signed half byte integer vector consists of 8 signed half byte integers contained in a doubleword. Each signed half byte integer element has a range from -8 to 7 with the sign on bit 3. This numeric data type is only used by an immediate source operand of doubleword in an instruction. It cannot be used for the destination operand or a non-immediate source operand. Hardware converts the vector into an 8-element signed word vector by sign extension. This is illustrated in Converting a packed half-byte vector to a 128-bit signed integer vector.

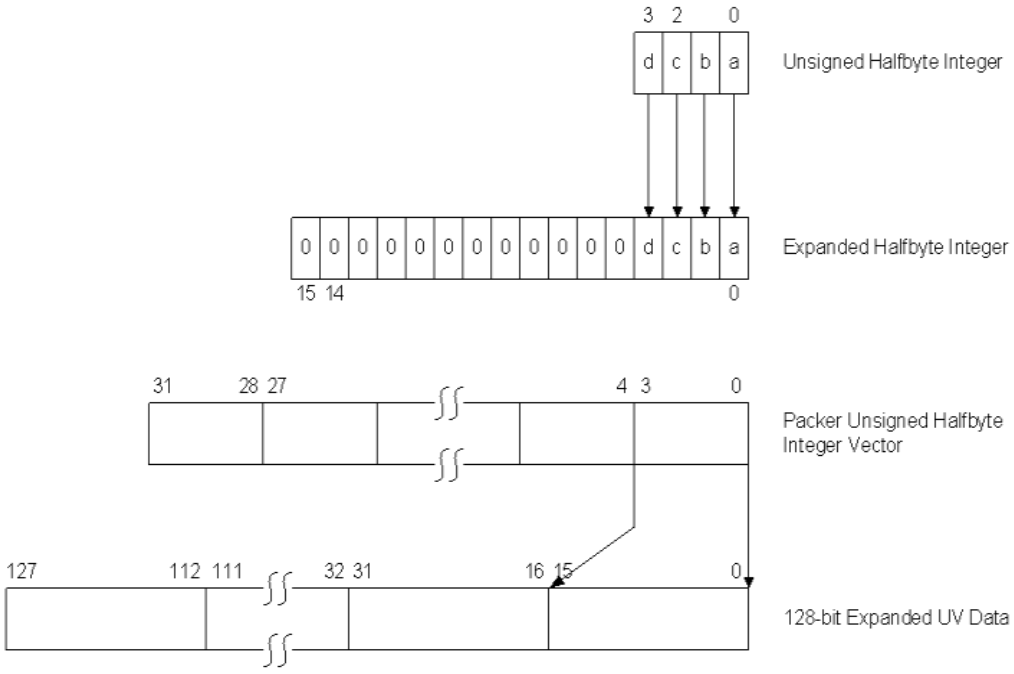
The shorthand format notation for a packed signed half-byte vector is **V**.

Converting a Packed Half-Byte Vector to a 128-bit Signed Integer Vector



Packed UnSigned Half-Byte Integer Vector

A packed unsigned half byte integer vector consists of 8 unsigned half byte integers contained in a doubleword. Each unsigned half byte integer element has a range from 0 to 15. This numeric data type is only used by an immediate source operand of doubleword in an instruction. It cannot be used for the destination operand or a non-immediate source operand. Hardware converts the vector into an 8-element signed word vector.

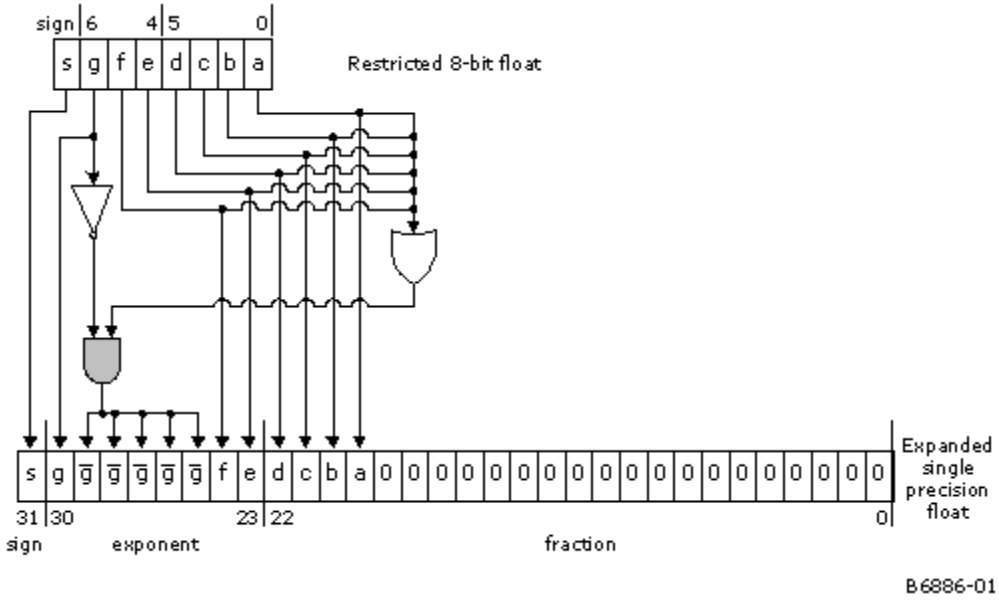


Packed Restricted Float Vector

A packed restricted float vector consists of 4 8-bit restricted floats contained in a doubleword. Each restricted float has the sign at bit 7, a 3-bit coded exponent in bits 4 to 6, a 4-bit fraction in bits 0 to 3, and an implied integer 1. The exponent is in excess-3 format - having a bias of 3. Restricted float provides zero, positive/negative normalized numbers with a small range (3-bit exponent) and small precision (4-bit fraction). This numeric data type is only used by an immediate source operand of doubleword in an instruction. It cannot be used for the destination operand, or a non-immediate source operand.

The following figure shows how to convert an 8-bit restricted float into a single precision float. Converting a 3-bit exponent with a bias of 3 to an 8-bit exponent with a bias of 127 is by adding 4, or equivalently copying bit 2 to bit 7 and putting the inverted bit 2 to bits 6:2. A special logic is also needed to take care of positive/negative zeros.

Conversion from a Restricted 8-bit Float to a Single-Precision Float



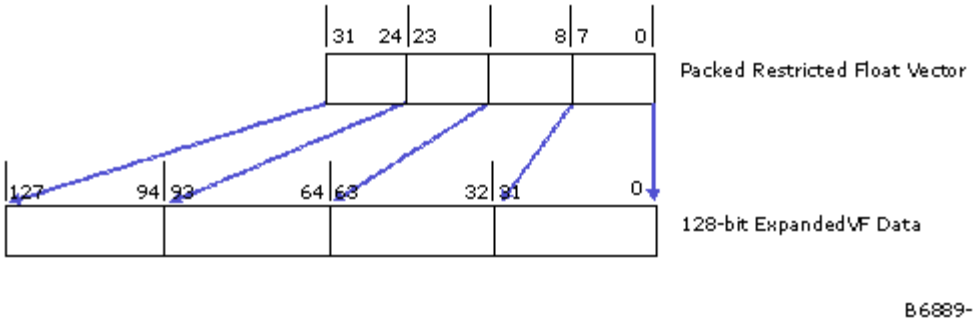
The following table shows all possible numbers of the restricted 8-bit float. Only normalized float numbers can be represented, including positive and negative zero, and positive and negative finite numbers. Normalized infinites, NaN, and denormalized float numbers cannot be represented by this type. It should be noted that this 8-bit floating point format does not follow IEEE-754 convention in describing numbers with small magnitudes. Specifically, when the exponent field is zero and the fraction field is not zero, an implied one is still present instead of taking a denormalized form (without an implied one). This results in a simple implementation but with a smaller dynamic range - the magnitude of the smallest non-zero number is 0.1328125.

Examples of Restricted 8-bit Float Numbers

Class	Hex #	Sign [7]	Exponent [6:4]	Fraction [3:0]	Extended 8-bit Exponent	Floating Number in Decimal
Positive Normalized Float	0x70-0x7F	0	111	0000 ... 1111	1000 0011	16 ... 31
	0x60-0x6F	0	110	0000 ... 1111	1000 0010	8 ... 15.5
	0x50-0x5F	0	101	0000 ... 1111	1000 0001	4 ... 7.75
	0x40-0x4F	0	100	0000 ... 1111	1000 0000	2 ... 3.875
	0x30-0x3F	0	011	0000 ... 1111	0111 1111	1 ... 1.9375
	0x20-0x2F	0	010	0000 ... 1111	0111 1110	0.5 ... 0.96875
	0x10-0x1F	0	001	0000 ... 1111	0111 1101	0.25 ... 0.484375
	0x01-0x0F	0	000	0001 ... 1111	0111 1100	0.1328125 ... 0.2421875
	0x00	0	000	0000	0000 0000	0 (+zero)
Negative Normalized Float	0xF0-0xFF	1	111	0000 ... 1111	1000 0011	-16 ... -31
	0xE0-0xEF	1	110	0000 ... 1111	1000 0010	-8 ... -15.5
	0xD0-0xDF	1	101	0000 ... 1111	1000 0001	-4 ... -7.75
	0xC0-0xCF	1	100	0000 ... 1111	1000 0000	-2 ... -3.875
	0xB0-0xBF	1	011	0000 ... 1111	0111 1111	-1 ... -1.9375
	0xA0-0xAF	1	010	0000 ... 1111	0111 1110	-0.5 ... -0.96875
	0x90-0x9F	1	001	0000 ... 1111	0111 1101	-0.25 ... -0.484375
	0x81-0x8F	1	000	0001 ... 1111	0111 1100	-0.1328125 ... -0.2421875
	0x80	1	000	0000	0000 0000	-0 (-zero)

The following figure shows the conversion of a packed exponent-only float to a 4-element vector of single precision floats.

The shorthand format notation for a packed signed half-byte vector is VF.



Floating Point Modes

Architecture supports two floating point operation modes, namely IEEE floating point mode (IEEE mode) and alternative floating-point mode (ALT mode). Both modes follow mostly the requirements in IEEE-754 but with different deviations. The deviations will be described in detail in later sections. The primary difference between these modes is on the handling of Infs, NaNs and denorms. The IEEE floating point mode may be used to support newer versions of 3D graphics API Shaders and the alternative floating-point mode may be used to support early Shader versions. Taking DirectX 3D graphics API Shaders for example, shader models before version 3.0 may use the alternative floating-point mode, while version 3.0 and following shader models may use the IEEE floating point mode.

These two modes are supported by all units that perform floating point computations, including execution units, shared functions like Extended Math and the Render Cache color calculator, and fixed functions like VF, Clipper, SF and WIZ. Host software sets floating point mode through the fixed function state descriptors for 3D pipeline and the interface descriptor for media pipeline. Therefore, different modes may be associated with different threads running concurrently. Floating point mode control for EU and shared functions are based on the floating-point mode field (bit 0) of *cr0* register.

These two modes are supported by all units that perform floating point computations, including execution units, shared functions like Extended Math, the Sampler and the Render Cache color calculator, and fixed functions like VF, Clipper, SF and WIZ. Host software sets floating point mode through the fixed function state descriptors for 3D pipeline and the interface descriptor for media pipeline. Therefore, different modes may be associated with different threads running concurrently. Floating point mode control for EU and shared functions are based on the floating-point mode field (bit 0) of *cr0* register.

IEEE Floating Point Mode

Partial Listing of Honored IEEE-754 Rules

Here is a summary of expected 32-bit floating point behaviors in architecture. Refer to IEEE-754 for topics not mentioned.

- INF - INF = NaN
- 0 * (+/-)INF = NaN
- 1 / (+INF) = +0 and 1 / (-INF) = -0
    - (+/-)INF / (+/-)INF = NaN as A/B = A * (1/B)

- INV (+0) = RSQ (+0) = +INF, INV (-0) = RSQ (-0) = -INF, and SQRT (-0) = -0
- RSQ (-finite) = SQRT (-finite) = NaN
- LOG (+0) = LOG (-0) = -INF, LOG (-finite) = LOG (-INF) = NaN
- NaN (any OP) any-value = NaN with one exception for min/max mentioned below. Resulting NaN may have different bit pattern than the source NaN.
- Normal comparison with conditional modifier of EQ, GT, GE, LT, LE, when either or both operands is NaN, returns FALSE. Normal comparison of NE, when either or both operands is NaN, returns TRUE.
    - **Note:** Normal comparison is either a **cmp** instruction or an instruction with conditional modifier
- Special comparison **cmpn** with conditional modifier of EQ, GT, GE, LT, LE, when the second source operand is NaN, returns TRUE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns FALSE. **Cmpn** of NE, when the second source operand is NaN, returns FALSE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns TRUE.
    - This is used to support the proposed IEEE-754R rule on **min** or **max** operations. For which, if only one operand is NaN, min and max operations return the other operand as the result.
- Both normal and special comparisons of any non-NaN value against +/- INF return exact result according to the conditional modifier. This is because that infinities are exact representation in the sense that +INF = +INF and -INF = -INF.
    - NaN is unordered in the sense that NaN != NaN.
- IEEE-754 requires floating point operations to produce a result that is the nearest representable value to an infinitely precise result, known as "round to nearest even" (RTNE). 32-bit floating point operations must produce a result that is within 0.5 Unit-Last-Place (0.5 ULP) of the infinitely precise result. This applies to addition, subtraction, and multiplication.
- All arithmetic floating point instructions does Round To Nearest Even at the end of the computation, except the round instructions.

Complete Listing of Deviations or Additional Requirements vs IEEE-754

For a result that cannot be represented precisely by the floating point format, the EU uses rounding to nearest or even to produce a result that is within 0.5 Unit-Last-Place(0.5 ULP) of the infinitely precise result.

Description
The rounding mode is specified by the Rounding Mode field in the Control Register.
Handle denorms as follows: • When inputs are denorms in mixed mode (one of the source operand is half float and other is single precision float OR source is half float and destination is single precision float), on upconversion, they are non-denorms. So computes in 32b can handle this. • When outputs are denorms, the denorms are retained if there is a format conversion.

Description
• Mixed mode operations should always behave like sum of individual operations.
Handle Mixed mode with Bfloat16 as follows: • When an input is bfloat16, it is upconverted to float32 before compute. • RNDZ is performed instead of the Rounding mode specified in Control Register. • Denorms are always retained during upconversion and downconversion.

Other information regarding floating-point behaviors:

- NaN input to an operation always produces NaN on output, however the exact bit pattern of the NaN is not required to stay the same (unless the operation is a raw "mov" instruction which does not alter data at all.)
- x*1.0f must always result in x (except denorm flushed and possible bit pattern change for NaN).
- x +/- 0.0f must always result in x (except denorm flushed and possible bit pattern change for NaN). But -0 + 0 = +0.
- Fused operations (such as *mac*, *dp4*, *dp3*, etc.) may produce intermediate results out of 32-bit float range, but whose final results would be within 32-bit float range if intermediate results were kept at greater precision. In this case, implementations are permitted to produce either the correct result, or else +/-inf. Thus, compatibility between a fused operation, such as *mac*, with the unfused equivalent, *mul* followed by *add* in this case, is not guaranteed.
- As the accumulator registers have more precision than 32-bit float, any instruction with accumulator as a source/destination operand may produce a different result than that using more general registers, as indicated in this table:
- Such an instruction may produce a different result than that using GRF registers.
- API Shader divide operations are implemented as x*(1.0f/y). With the two-step method, x*(1.0f/y), the multiply and the divide each independently operate at the 32-bit floating point precision level (accuracy to 1 ULP).
- See the Type Conversion section for rules on converting to and from float representations.

Min Max of Floating Point Numbers

A special comparison called Compare-NaN is introduced in the architecture to handle the difference of above-mentioned floating-point comparison and the rules on supporting MIN/MAX. To compute the MIN or MAX of two floating-point numbers, if one of the numbers is NaN and the other is not, MIN or MAX of the two numbers returns the one that is not NaN. When two numbers are NaN, MIN or MAX of the two numbers returns source1.

When both the sources are NaN inputs, the special case in the section 2.4.8 Floating point Min/Max Operations describe the results.

When one source is SNAN, DX,OCL and IEEE treat the outputs differently. The special case section 2.4.8 Floating point Min/Max operations described the results.

Min and Max is supported by conditional select.

Note even though f0.0 is specified in the instruction, the flag register is not touched by this instruction.

The following tables detail the rules for this special compare-NaN operation for floating-point numbers. Notice that excepting "Not-Equal-To" comparison-NaN, last columns in all other tables have 'T'.

Results of "Less-Than" Comparison-NaN - CMPN.L

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	F	T	T	T	T	T	T	T	T
-Fin	F	T/F	T	T	T	T	T	T	T
-denorm	F	F	F	F	F	F	T	T	T
-0	F	F	F	F	T	F	T	T	T
+0	F	F	F	F	F	F	T	T	T
+denorm	F	F	F	F	F	F	T	T	T
+Fin	F	F	F	F	F	F	T/F	T	T
+inf	F	F	F	F	F	F	F	F	T
NaN	F	F	F	F	F	F	F	F	F

Results of "Greater-Than or Equal-To" Comparison-NaN - CMPN.GE

src0 src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf	T	F	F	F	F	F	F	F	T
-Fin	T	T/F	F	F	F	F	F	F	T
-denorm	T	T	T	T	T	T	F	F	T
-0	T	T	T	T	F	T	F	F	T
+0	T	T	T	T	T	T	F	F	T
+denorm	T	T	T	T	T	T	F	F	T
+Fin	T	T	T	T	T	T	T/F	F	T
+inf	T	T	T	T	T	T	T	T	T
NaN	F	F	F	F	F	F	F	F	F

Alternative Floating Point Mode

The key characteristics of the alternative floating-point mode is that NaN, Inf, and denorm are not expected for an application to pass into the graphics pipeline, and the graphics hardware must not generate NaN, Inf, or denorm as a single precision result in destination of a computation/conversion operation. For example, a result that is larger than the maximum representable floating-point number is expected to be flushed to the largest representable floating-point number, i.e., +fmax. The fmax has an exponent of 0xFE and a mantissa of all one's, which is the same for IEEE floating point mode.

ALT mode is ignored by systolic pipeline instructions.

ALT mode must not be used with bfloat datatypes in source or destination.

Here is the complete list of the differences of legacy graphics mode from the relaxed IEEE-754 floating point mode.

- Any +/- INF result must be flushed to +/- fmax, instead of being output as +/- INF.
- Extended mathematics functions of log(), rsq(), and sqrt() take the absolute value of the sources before computation to avoid generating INF and NaN results.

Supported Legacy Float Mode and Impacted Units shows the support of these differences in various hardware units.

Supported Legacy Float Mode and Impacted Units

IEEE-754 Deviations	VF	Clipper	SF	WIZ	EU	EM	Sampler	RC
Any +/- INF result flushed to +/- fmax	Y	Y	Y	Y	Y	Y	Y	Y
Log, rsq, sqrt take abs() of sources	N/A	N/A	N/A	N/A	N/A	Y	N/A	N/A

Supported Legacy Float Mode and Impacted Units

IEEE-754 Deviations	VF	Clipper	SF	WIZ	EU	EM	RC
Any +/- INF result flushed to +/- fmax	Y	Y	Y	Y	Y	Y	Y
Log, rsq, sqrt take abs() of sources	N/A	N/A	N/A	N/A	N/A	Y	N/A

Dismissed legacy behaviors shows some of the desired or recommended alternative floating point mode behaviors that do not have hardware design impact. The reasons of not needing special hardware support for these items are also provided. This is based on the compliance requirement that can be found in the DirectX 9 specification: "**Handling of NaNs, Infs, and denorms is undefined. Applications should not pass in such values into the graphics pipeline.**"

Dismissed Legacy Behaviors

Suggested IEEE-754 Deviations	Reason for Dismiss
Mov forces (+/-)INF to (+/-)fmax	(+/-)INF is never present as input
(+/-)INF - (+/-)INF = +/- fmax instead of NaN	(+/-)INF is never present as input
Denorm must be flushed to zero in all cases (including trivial mov and point sampling)	Denorm is never present as input
Anything*0=0 (including NaN*0=0 and INF*0=0)	NaN and INF are never present as input
Except propagated NaN, NaN is never generated	NaN is never present as input and GFX never generates NaN based on rules in the previous table
An input NaN gets propagated excepting (a)-(d)	NaN is never present as input
(a) Rcp (and rsq) of 0 yields fmax	N/A, as it is already covered by the general rule "Any +/- INF result flushed to +/- fmax"

Suggested IEEE-754 Deviations	Reason for Dismiss
(b) Sampler honors 0/0 = 0 as if (1/0)*0	There is no divide in Sampler
I Rcp (and rsq) of INF yields +/- 0	(+/-)INF is never present as input
(d) Sampler honors INF/INF = 0 as if (1/INF)=0 followed by Anything*0 = 0	There is no divide in Sampler

Floating-Point Support

The following sections describe the floating-point support relative to the IEEE Standard for Floating-Point Arithmetic, currently IEEE Standard 754-2008. These sections cover binary floating-point arithmetic, as the EU provides no support for decimal floating-point arithmetic.

Note: Hardware alone is usually not fully conformant to the IEEE standard. It requires software functions to supplement the hardware.

The following has been added:

- Adds the HF (Half Float) type and a corresponding HF execution data type and execution path.
- Adds flags to indicate IEEE floating-point exceptions and to enable or disable exception reporting to those flags.
- Adds the Single Precision Denorm Mode bit in Control Register 0. It can be enabled to allow calculations using the F (Float) type in IEEE floating-point mode to support denormals and gradual underflow.

Floating-Point Types and Values

The EU supports 16-bit (HF, Half Float), 32-bit (F, Float), and 64-bit (DF, Double Float) types in the IEEE Standard formats (respectively *binary16*, *binary32*, and *binary64* in IEEE 754-2008). See Floating-Point Numeric Data Types for the layout of the supported floating-point types.

Any bit pattern for a floating-point value corresponds to a value defined by the standard: +/- finite (normalized nonzero finite number), +/- 0 (signed zero), +/- inf (signed infinity), +/- denorm (denormalized, very small but nonzero number), or NaN (Not a Number). A NaN can be a signaling NaN (sNaN) or quiet NaN (qNaN).

These operating modes are available for the different floating-point types:

- Half Float uses the IEEE floating-point mode.
- Half Float support for denormals and gradual underflow is controlled by the Half Precision Denorm Mode bit.
- Float can use the ALT (Alternative Floating-Point Mode) or the IEEE floating-point mode. In IEEE mode, support for denormals and gradual underflow is controlled by the Single Precision Denorm Mode bit in the Control Register.
- Double Float uses the IEEE floating-point mode. Support for denormals and gradual underflow is controlled by the Double Precision Denorm Mode bit in the Control Register.

Flush to zero is not defined by IEEE Standard 754 but is implementation-specific and required by some APIs (including DirectX), thus the EU ISA supports either using or flushing Float or Double Float denorms based on the respective Denorm Mode bits.

Specifications in this volume sometimes reference +/-fmax, the largest finite magnitude representable in a format, and +/-fmin, the smallest normalized nonzero magnitude representable in a format. Calculating those values uses the extreme exponent values for finite nonzero floating-point values, Emax and Emin below, along with the number of explicit fraction bits (not counting the implicit bit in the significand). The following table provides these values, with the fmax and fmin values generally approximate.

Floating-Point Type Parameters

Type	Exp. Bits	Exp. Bias	Emax	Emin	Explicit Fraction Bits	fmax	fmin
HF	5	15	15	-14	10	65504.0	about 6.1E-5
F	8	127	127	-126	23	about 3.4E38	about 1.18E-38
DF	11	1023	1023	-1022	52	about 1.79E308	about 2.23E-308

Where f is the number of explicit fraction bits, the general formula for fmax is $(2.0 - 2^{-f}) * 2^{Emax}$.

The general formula for fmin is 2^{Emin}.

Not a Number (NaN) Formats

A NaN has a biased exponent field with all bits set (as if encoding an exponent of Emax + 1), a nonzero fraction field (as a zero-fraction field with that exponent indicates infinity), and either sign bit.

As specified in IEEE Standard 754-2008, the MSB of the fraction field, what would be the first bit following the binary point in a numeric value, determines a NaN's type:

0 - Signaling NaN (sNaN). The remaining fraction bits cannot all be zero.

1 - Quiet NaN (qNaN). The remaining fraction bits can have any value.

When an sNaN is an input, an operation normally signals the Invalid Operation exception and *quietizes* the NaN, producing the equivalent qNaN value, with MSB set to 1, at the output. Raw moves do not check for NaNs and do not signal exceptions or quietize NaN values.

When QNAN as one of the inputs to an operation this results in QNAN without raising the exception flag. This silently propagates and the output is the same QNAN as in the input.

Intel specifies the value *qNaN Indefinite* as a quiet NaN with all zeros in the remaining fraction bits, those other than the MSB. This value is useful because it is never produced by quietizing an sNaN, thus qNaN Indefinite may be used to initialize floating-point values that are not otherwise initialized by software, allowing the uninitialized case to be distinguished.

The EU applies numeric source modifiers (**-**, **(abs)**, or **-(abs)**) to NaN source values as well as to other values, possibly changing the sign bit of a NaN value when it is propagated. NaN sign bits are normally don't care values.

Per IEEE Standard 754-2008, a NaN's *payload* is contained in all fraction field bits other than the fraction MSB. Thus in the overall floating-point format, the sign bit, biased exponent, and fraction MSB are not part of the payload. NaN payload values are not affected by quietizing or by source modifiers. As noted above, an sNaN must have a nonzero payload and a qNaN can have any payload.

Floating-Point Rounding Modes

The EU supports the four rounding modes required in IEEE Standard 754 for binary floating-point arithmetic. If the unrounded result of infinite precision and range is exactly representable in the destination format, then that exact result is produced, and no exception is signaled. For an unrounded result of infinite precision and range that is not exactly representable in the destination format (an *inexact* result), rounding chooses a numerically adjacent value in the destination format, while signaling the Inexact, Overflow, or Underflow exceptions when appropriate. The four rounding modes are:

RNE - Round to nearest or even. Choose the value in the destination precision nearest to the unrounded result. If the unrounded result is midway between two such values, choose the value with its least significant fraction bit as 0 (*even*).

RD - Round down, toward minus infinity.

RU - Round up, toward plus infinity.

RZ - Round toward zero.

The rounding mode is specified by the Rounding Mode field in the Control Register. It is initialized by Thread Dispatch. The normal default value is round to nearest or even. The Rounding Mode can be read to check the mode and written to change it. The Control Register and the Rounding Mode value are thread-specific; the Rounding Mode applies to all floating-point types, all execution channels, and all floating-point instructions executed by the thread after it is assigned.

Rounding an inexact result signals the Inexact Exception.

Rounding an inexact result preserves the sign of the result.

Infinities and NaNs are exact results and are not affected by the rounding mode.

Zeros are exact results, but the signs of zero results are affected by the rounding mode in certain cases:

X - X = +0 for RNE, RU, RZ

X - X = -0 for RD

(+0) + (-0) = (-0) + (+0) = +0 for RNE, RU, RZ

(+0) + (-0) = (-0) + (+0) = -0 for RD

Regardless of the rounding mode, (+0) + (+0) = +0 and (-0) + (-0) = -0.

The *directed rounding modes* are round down, round up, and round toward zero.

In IEEE mode, when a floating-point overflow occurs the result is determined by the sign of the result and the rounding mode:

+ and (RNE or RU): + inf

+ and (RD or RZ): + fmax

- and (RU or RZ): - fmax

- and (RNE or RD): - inf

Note that for floating-point overflow in IEEE mode, RNE always produces +/- inf and RZ always produces +/- fmax.

Floating-Point Operations and Precision

IEEE Standard 754-2008 requires the following floating-point operations to be precise within <= 0.5 *ulp* (unit in the last place) when using the round to nearest or even rounding mode:

ADD (*add* instruction)

DIV (*math* instruction with FDIV function code)

FMA (fused multiply add, *mad* instruction)

MUL (*mul* instruction)

SQRT (*math* instruction with SQRT function code)

SUB (*add* instruction using one **-** source modifier)

Conversions (float to float, float to int, and int to float)

Min/Max

Compare

Single Precision Floating-Point Rounding to Integral Values

The **rndd** (Round Down), **rnde** (Round to Nearest or Even), **rndu** (Round Up), and the **rndz** (Round to Zero) instructions round arbitary Float values to integral Float values. Each instruction specifies its rounding mode so these instructions are not affected by the Rounding Mode in the Control Register.

An integral source value produces the same value for the destination (ignoring any saturation). For magnitudes >= 8,388,608 (2^{23}) all Float values are integral.

The rounding instructions are sign preserving.

Signed zeros are propagated. In IEEE mode, signed infinities are propagated. In IEEE mode, sNaN inputs are quietized, the equivalent qNaN is produced, and the Invalid Operation exception is indicated. In IEEE mode, qNaN inputs are propagated.

No other exceptions are signaled for these instructions. For example, if the source and result values differ, the Inexact exception is not signaled.

The Single Precision Denorm Mode in the Control Register affects the results of the **rndd** and **rndu** instructions for denorm source values.

Floating-Point to Integer Conversion

The *mov* and *sel* instructions can be used to convert floating-point values to integers. In the tables below, Imin is the smallest representable value in a signed integer type, Imax is the largest representable value in an integer type, and f is a finite floating-point value after rounding to an integral value using the current rounding mode.

Converting unrepresentable floating-point values, including infinities, NaNs, and values that convert to integers outside of the destination type's range, signal Invalid Operation exceptions. When the destination integer type is unsigned, normalized nonzero negative inputs signal Invalid Operation exceptions and negative denorm inputs signal Inexact exceptions.

Data written in accumulator using implicit or explicit destination will not be IEEE compliant.

Floating-Point to Integer Conversion Results and Exceptions for Signed Integer Types

FP Value	Integer Result	FP Exception
qNaN	0	Invalid Operation
sNaN	0	Invalid Operation
+inf	Imax	Invalid Operation
f > Imax	Imax	Invalid Operation
Imin <= f <= Imax	f	Inexact if rounding changed f
f < Imin	Imin	Invalid Operation
-inf	Imin	Invalid Operation

Floating-Point to Integer Conversion Results and Exceptions for Unsigned Integer Types

FP Value	Integer Result	FP Exception
qNaN	0	Invalid Operation
sNaN	0	Invalid Operation
+inf	Imax	Invalid Operation
f > Imax	Imax	Invalid Operation
0 <= f <= Imax	f	Inexact if rounding changed f
f = -0	0	no exception
-1 < f < -0	0	Inexact
-1 < f < -0	Real Indefinite	Invalid Operation
-fmax <= f <= -1	Real Indefinite	Invalid Operation
-inf	Real Indefinite	Invalid Operation

Note: Real Indefinite is encoded as Integer value 0.

Integer to Floating-Point Conversion

Integer to floating-point conversion follows these rules in IEEE mode:

- The result is never +/- inf, never NaN, and never -0.
- If the integer source value is not exactly representable in the destination floating-point format, use the current rounding mode to choose an adjacent floating-point value and signal the Inexact Exception.
- If the integer source value is too large to represent in the destination floating-point format (only possible when converting to Half Float from D, UD, or UW) then signal the Overflow Exception. Based on the sign and the current rounding mode, the result is +/- fmax or +/- inf, as described in the Overflow Exception section.

Floating-Point Min/Max Operations

In the following Min/Max operations, sNaN inputs are preferred to non-NaN inputs and non-NaN inputs are preferred to qNaN inputs.

Min(x, qNaN) = Min(qNaN, x) = x with no exceptions signaled.

Min(x, sNaN) = Min(sNaN, x) = qNaN (quietized value corresponding to the input sNaN) and signal the Invalid Operation exception.

Note: DX deviates from this rule:

The DX behavior is inferred from the denorm bit.

Min(x, sNaN) = Min(sNaN, x) = x

Max(x, qNaN) = Max(qNaN, x) = x with no exceptions signaled.

Max(x, sNaN) = Max(sNaN, x) = qNaN (quietized value corresponding to the input sNaN) and signal the Invalid Operation exception.

Note: DX deviates from this rule:

The DX behavior is inferred from the denorm bit.

Max(x, sNaN) = Max(sNaN, x) = x

Special cases(when both sources are NaN inputs)

Min(qNaN, qNan) = qNaN (of the first source) and no exception raised

Min(qNaN, sNaN) = Min(sNaN, qNaN) = qNaN (quiet-ized sNaN and signal the Invalid Operation exception)

Min(sNaN, sNaN) = qNaN(of the first source and signal Invalid Operation Exception)

Max(qNaN, qNan) = qNaN (of the first source) and no exception raised

Max(qNaN, sNaN) = Max(sNaN, qNaN) = qNaN (quiet-ized sNaN and signal the Invalid Operation exception)

Max(sNaN, sNaN) = qNaN(of the first source and signal Invalid Operation Exception)

IEEE Floating-Point Exceptions

The EU detects the five floating-point exceptions defined by IEEE Standard 754:

Invalid Operation

Division by Zero

Overflow

Underflow

Inexact

Signaling Floating-Point Exceptions

When enabled in the Control Register, floating-point exceptions are detected and set *sticky* flag bits in the State Register. There is no mechanism for automatic transfer to a handler, so floating-point exceptions are not handled like other exceptions described in the Exceptions chapter.

Setting flags to indicating Floating-Point exceptions is the default exception handling approach specified by IEEE Standard 754. The flag bits are sticky because in normal operation the EU only sets these bits as exceptions occur, and does not clear these bits, so a set value sticks until cleared by software. The Control Register and State Register are cleared at reset and initialized at thread load. Both are read/write registers. These fields are used:

- **IEEE Exception Trap Enable** (Control Register cr0.0:ud bit 9). This bit enables trapping IEEE exception flags. This control bit may be updated by software. It is initially zero on thread load. If enabled, IEEE floating-point exceptions set sticky bits in the IEEE Exception field of sr0.1, in the State Register. **Note:** Software must set this flag at thread start to use the IEEE Exception flags.
- **IEEE Exception**. (State Register sr0.1:ud bits 7:0). The IEEE exception bits are sticky bits set by the opcodes when floating-point exceptions are triggered. These bits are defined per thread and all channels update one set of sticky bits. These bits are cleared on thread load and may be cleared by software. Exception updates to these bits may be disabled by clearing the IEEE Exception Trap Enable bit in the Control Register. The following table specifies the IEEE exception bits:

Bits	Definition
7:5	Reserved
4	Inexact Exception
3	Overflow Exception
2	Underflow Exception
1	Division by Zero Exception
0	Invalid Operation Exception

The IEEE exception flags are not generated for the following cases:

Operations involving non IEEE float formats (e.g. bfloat16, bfloat8, tfloat32 etc.) does not trigger these exceptions.

When output is floating point data type, and corresponding data type denormal bit is set flush to zero, no flags are generated.

For floating point to integer conversion, when the corresponding floating point data type denomormal bit is set flush to zero or IEEE Float to Integer Rounding Mode is set to zero (bit 12 of CR0.DW0), no flags are generated.

Invalid Operation Exception

An Invalid Operation exception is signaled by any operation on a signaling NaN (sNaN) or by certain combinations of operations and operands with undefined results, always producing a quiet NaN (qNaN) result.

The following specific operations signal Invalid Operation, where *x* is a positive, finite, nonzero, and normalized number:

+inf - (+inf) or (-inf) - (-inf)

+/- 0 / +/- 0

+/- inf / +/- inf

+/- 0 x +/- inf or +/- inf x +/- 0

Remainder(+/- *x*, +/- 0)

Remainder(+/- inf, +/- *x*)

Sqrt(-*x*)

Note that Sqrt(-0) is -0 per IEEE Standard 754 and does not cause any exception.

These instructions can signal specific Invalid Operation exceptions (and also on sNaN inputs except for Float inputs in ALT mode), producing a qNaN result:

add

dp2, *dp3*, *dp4*, and *dph* (the Dot Product instructions)

line

lrp

mac

mad

madm

math with the FDIV, SQRT, or RSQRTM function codes

mul

pln

These other instructions signal Invalid Operation exceptions on sNaN inputs (except for Float inputs in ALT mode), producing a qNaN result:

cmp, *cmpn*

frc

math with all other function codes for floating-point operations

mov except for raw moves

rndd, *rnde*, *rndu*, and *rndz* (the Round instructions)

sel

smov except for raw moves

Division by Zero Exception

The operation +/- *x* / +/- 0, where *x* is a positive, finite, nonzero, and normalized number, signals the Division by Zero Exception and produces a correctly signed +/- inf result. Note that in accordance with the standard, +/- 0 / +/- 0 signals Invalid Operation, does not signal Division by Zero, and produces a qNaN result. This behavior can occur for the *math* instruction with the FDIV function code.

The operation LOG_2(+/- 0) signals the Division by Zero Exception and produces -inf as the result. This behavior can occur for the *math* instruction with the LOG function code.

Overflow Exception

A floating-point *overflow* occurs when an operation with a finite result produces an internal result with magnitude > fmax, the maximum representable finite value in the destination format. The internal result is rounded to the destination precision with the current rounding mode but has an unbounded exponent. An overflow produces +/- inf or +/- fmax as the result, depending on the sign and the rounding mode.

The following algorithm describes floating-point overflow processing:

1. Compute the result with infinite precision and unbounded range.
2. Normalize the result using an unbounded exponent.
3. Round the result to the destination precision using an unbounded exponent and the current rounding mode.
4. If (abs(rounded unbounded result) > fmax(destination format)) {

    Set the Overflow Exception flag to 1.

    Output = +/- inf or +/- fmax depending on the sign and the rounding mode:

        + and (RNE or RU): + inf

        + and (RD or RZ): + fmax

        - and (RU or RZ): - fmax

- and (RNE or RD): - inf

}

}

Note: The first three steps of the overflow and underflow algorithms are identical.

Underflow Exception

An *underflow* occurs when an operation produces a tiny but nonzero inexact result *x*, with abs(*x*) < fmin, where fmin = 2^{Emin}. See the Floating-Point Type Parameters table for the fmin and Emin values for different floating-point types.

IEEE Standard 754 allows underflow to be determined before or after rounding. The Execution Unit determines underflow after rounding, which is consistent with the behavior of the x87 and SSE instructions in the CPU.

When denorms are flushed to zero, no underflow exceptions are signaled. Flush to zero is not defined by IEEE Standard 754 but is implementation specific and required by some APIs (including DirectX), thus the EU ISA supports either using or flushing Float or Double Float denorms based on the respective Denorm Mode bits.

When denorms are enabled, if an operation's internal result rounded to the destination precision, but using an unbounded exponent range, has a magnitude that is less than fmin but nonzero AND the denorm result in the destination format is inexact, then signal the Underflow Exception.

The rounded result can be +/- 0, +/- fmin, or (when denorms are enabled) +/- denorm.

IEEE Standard 754 requires the non-intuitive behavior that an exact denorm result does not set the Underflow Exception flag.

The following algorithm describes floating-point underflow processing:

1. Compute the result with infinite precision and unbounded range.
2. Normalize the result using an unbounded exponent.
3. Round the result to the destination precision using an unbounded exponent and the current rounding mode.
4. If (0 < abs(rounded unbounded result) < fmin(destination format) {

    if (flush denorms to zero) {

        Output = 0; // No underflow exception.

    }

    Else {

        Renormalize to the bounded exponent with the original infinite precision value...

            ...and round that value to the destination precision using the current rounding mode.

```
            If ( the just computed value differs from the value computed in step (3) in exponent or
            mantissa ) {
                Set the Underflow Exception flag to 1.
            } // Note: Underflow is not set if the tiny result is the same as when computed with an
            unbounded exponent.
            Output = rounded result using the destination precision and destination exponent range;
        }
    }
    Else { // Not a tiny number.
        Output = rounded number;
    }
```

Inexact Exception

An Inexact Exception occurs when the internal unrounded result, with infinite precision and unbounded exponent range, differs from the generated result after format conversion, normalizing or denormalizing, and rounding. An Inexact Exception occurs irrespective of any saturation to exact zero or exact +1.0. The Inexact Exception is normal and may occur more often than not. For example, the calculation 1.0 / 3.0 is inexact in any binary floating-point format. These rules determine whether a result is inexact:

- Infinities and NaNs are never inexact.
- Flushing a denorm internal result to zero (always for Half Float and if the appropriate Denorm Mode is 0 for Float and Double Float) is always inexact.
- If any rounding occurs using the current Rounding Mode, so the rounded result differs from the internal unrounded result, the result is inexact. However explicit round to integral using any of the rounding instructions (*rndd*, *rnde*, *rndu*, and *rndz*) is never inexact.

Floating-Point Compare Operations

Four mutually exclusive relations are possible between two floating-point values, src0 and src1:

Less than. src0 < src1 and neither source is NaN.

Equal. src0 = src1 and neither source is NaN.

Greater than. src0 > src1 and neither source is NaN.

Unordered. Any source is NaN.

Any NaN compares unordered to any value, including itself.

Infinities of the same sign compare as equal.

Zeros compare as equal regardless of sign: -0 = +0.

Floating-Point Compare Relations

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	E	L	L	L	L	L	L	L	U
-fin	G	*	L	L	L	L	L	L	U
-denorm	G	G	*^	L^	L^	L^	L	L	U
-0	G	G	G^	E	E	L^	L	L	U
+0	G	G	G^	E	E	L^	L	L	U
+denorm	G	G	G^	G^	G^	*^	L	L	U
+fin	G	G	G	G	G	G	*	L	U
+inf	G	G	G	G	G	G	G	E	U
NaN	U	U	U	U	U	U	U	U	U

Notes	
*	Relation can be L, E, or G.
^	When denorms are flushed to zero then all denorms and zeros compare as E.

The next six tables show the results of six specific comparisons, corresponding to the **.g**, **.l**, **.e**, **.ne**, **.ge**, and **.le** conditional modifiers used with the *cmp* instruction and a floating-point source type. Any NaN source produces a false comparison result for these modifiers other than **.ne** and produces a true comparison result for the **.ne** modifier.

Results of Greater Than Comparison -- *cmp*.g

src1 / src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
-denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	F	F	F^	F	F	F
+0	T	T	T^	F	F	F^	F	F	F
+denorm	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	F	F
NaN	F	F	F	F	F	F	F	F	F

Notes	
*	Result can be T or F.

src1 src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
-denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	F	F	F^	F	F	F
+0	T	T	T^	F	F	F^	F	F	F
+denorm	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	F	F
NaN	F	F	F	F	F	F	F	F	F
Notes									
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.g** comparison result F.								

Results of Less Than Comparison -- *cmp*.l

src1 src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	T	T	T	T	T	T	T	F
-fin	F	*	T	T	T	T	T	T	F
-denorm	F	F	*^	T^	T^	T^	T	T	F
-0	F	F	F^	F	F	T^	T	T	F
+0	F	F	F^	F	F	T^	T	T	F
+denorm	F	F	F^	F^	F^	*^	T	T	F
+fin	F	F	F	F	F	F	*	T	F
+inf	F	F	F	F	F	F	F	F	F
NaN	F	F	F	F	F	F	F	F	F
Notes									
*	Result can be T or F.								
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.l** comparison result F.								

Results of Equal Comparison -- *cmp*.e

src1 src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	T	F	F	F	F	F	F	F	F
-fin	F	*	F	F	F	F	F	F	F
-denorm	F	F	*^	F^	F^	F^	F	F	F
-0	F	F	F^	T	T	F^	F	F	F
+0	F	F	F^	T	T	F^	F	F	F
+denorm	F	F	F^	F^	F^	*^	F	F	F
+fin	F	F	F	F	F	F	*	F	F
+inf	F	F	F	F	F	F	F	T	F
NaN	F	F	F	F	F	F	F	F	F

Notes	
*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.e** comparison result T.

Results of Not Equal Comparison -- *cmp*.ne

src1 src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	F	T	T	T	T	T	T	T	T
-fin	T	*	T	T	T	T	T	T	T
-denorm	T	T	*^	T^	T^	T^	T	T	T
-0	T	T	T^	F	F	T^	T	T	T
+0	T	T	T^	F	F	T^	T	T	T
+denorm	T	T	T^	T^	T^	*^	T	T	T
+fin	T	T	T	T	T	T	*	T	T
+inf	T	T	T	T	T	T	T	F	T
NaN	T	T	T	T	T	T	T	T	T

Notes	
*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.ne** comparison result F.

Results of Greater Than or Equal Comparison -- *cmp*.ge

src1 src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	T	F	F	F	F	F	F	F	F
-fin	T	*	F	F	F	F	F	F	F
-denorm	T	T	*^	F^	F^	F^	F	F	F
-0	T	T	T^	T	T	F^	F	F	F
+0	T	T	T^	T	T	F^	F	F	F
+denorm	T	T	T^	T^	T^	*^	F	F	F
+fin	T	T	T	T	T	T	*	F	F
+inf	T	T	T	T	T	T	T	T	F
NaN	F	F	F	F	F	F	F	F	F

Notes

*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.ge** comparison result T.

Results of Less Than or Equal Comparison -- *cmp*.le

src1 src0	-inf	-fin	-denorm	-0	+0	+denorm	+fin	+inf	NaN
-inf	T	T	T	T	T	T	T	T	F
-fin	F	*	T	T	T	T	T	T	F
-denorm	F	F	*^	T^	T^	T^	T	T	F
-0	F	F	F^	T	T	T^	T	T	F
+0	F	F	F^	T	T	T^	T	T	F
+denorm	F	F	F^	F^	F^	*^	T	T	F
+fin	F	F	F	F	F	F	*	T	F
+inf	F	F	F	F	F	F	F	T	F
NaN	F	F	F	F	F	F	F	F	F

Notes

*	Result can be T or F.
^	When denorms are flushed to zero then all denorms and zeros compare as equal, making the **.le** comparison result T.

Type Conversion

Float to Integer

Converting from float to integer is based on rounding toward zero (RTZ is for DX, IEEE expects all four rounding modes). If the floating-point value is +0, -0, +Denorm, -Denorm, +NaN -r -NaN, the resulting integer value is always 0. If the floating-point value is positive infinity (or negative infinity), the conversion result takes the largest (or the smallest) represent-able integer value. If the floating-point value is larger (or smaller) than the largest (or the smallest) represent-able integer value, the conversion result takes the largest (or the smallest) represent-able integer value. The following table shows these special cases. The last two rows are just examples. They can be any number outside the represent-able range of the output integer type (UQ, Q, UD, D, UW, W, UB and B).

Input Format			Output Format					
F	UQ	Q	UD	D	UW	W	UB	B
+/- Zero	0000000000000000	0000000000000000	00000000	00000000	00000000	00000000	00000000	00000000
+/- Denorm	0000000000000000	0000000000000000	00000000	00000000	00000000	00000000	00000000	00000000
NAN	0000000000000000	0000000000000000	00000000	00000000	00000000	00000000	00000000	00000000
-NAN	0000000000000000	0000000000000000	00000000	00000000	00000000	00000000	00000000	00000000
INF	FFFFFFFFFFFFFFFF	7FFFFFFFFFFFFFFF	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-INF	0000000000000000	8000000000000000	00000000	80000000	00000000	00008000	00000000	00000080
$+2^{32}$ (*)	0000000100000000	0000000100000000	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
$-2^{32}-1$ (*)	0000000000000000	FFFFFFFEFFFFFFFF	00000000	80000000	00000000	00008000	00000000	00000080
$+2^{64}$ (*)	FFFFFFFFFFFFFFFF	7FFFFFFFFFFFFFFF	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
$-2^{64}-1$(*)	0000000000000000	8000000000000000	00000000	80000000	00000000	00008000	00000000	00000080

Integer to Integer with Same or Higher Precision

Converting an unsigned integer to a signed or an unsigned integer with higher precision is based on zero extension.

Converting an unsigned integer to a signed integer with the same precision is based on modular wrap-around. Without saturation, a larger than represent-able number becomes a negative number. With saturation, a larger than represent-able number is saturated to the largest positive represent-able number.

Converting a signed integer to a signed integer with higher precision is based on sign extension.

Converting a signed integer to an unsigned integer with higher precision is based on sign extension. Without saturation, a negative number becomes a large positive number with the sign bit wrapped-up. With saturation, a negative number is saturated to zero.

Integer to Integer with Lower Precision

Converting a signed or an unsigned integer to a signed or an unsigned integer with lower precision is based on bit truncation. Without saturation, only the lower bits are kept in the output regardless of the sign-ness of input and output. With saturation, a number that is outside the represent-able range is saturated to the closest represent-able value.

Integer to Float

Converting a signed or an unsigned integer to a single precision float number is to round to the closest representable float number. For any integer number with magnitude less than or equal to 24 bits, resulting float number is a precise representation of the input. However, if it is more than 24 bits, by default a "round to nearest even" is performed.

Double Precision Float to Single Precision Float

Double Precision Float	Single Precision Float
-inf	-inf
-finite	-finite/-denorm/-0
-denorm	-0
-0	-0
+0	+0
+denorm	+0
+finite	+finite/+denorm/+0
+inf	+inf
NaN	NaN

The upper Dword of every Qword will be written with undefined value when converting DF to F.

Single Precision Float to Double Precision Float

Converting a single precision floating-point number to a double precision floating-point number will produce a precise representation of the input.

Single Precision Float	Double Precision Float
-inf	-inf
-finite	-finite
-denorm	-finite
-0	-0
+0	+0
+denorm	+finite
+finite	+finite
+inf	+inf
NaN	NaN

Exceptions

The Architecture defines a basic exception handling mechanism for several exception cases. This mechanism supports both normal operations such as extensions of the mask-stack depth, as well as detecting some illegal conditions.

Exception Types

Type	Trigger / Source	Sync/Async Recognition
Software Exception	Thread code	Synchronous
Breakpoint	• A bit in the instruction word • Breakpoint IP match • Breakpoint Opcode match	Synchronous
Illegal Opcode	Hardware	Synchronous
Halt	MMIO register write	Asynchronous
Context Save/Restore	Preemption Interrupt	Asynchronous

Threads may choose which exceptions to recognize and which to ignore. This mask information is specified on a per-kernel basis in fixed function state generated by the driver and delivered to the EU as part of a new thread dispatch. Upon arrival at the EU, the exception-mask information is used to initialize the exception enable fields of that thread's cr0.1 register, which controls exception recognition. This register is instantiated on a per-thread basis, allowing independent control of exception type recognition across hardware threads. The exception enables bits in the cr0.1 register are read/write, and thus can be enabled/disabled via software at any time during thread execution.

The exception handling mechanism relies on the System Routine, a single subroutine that provides common exception handling for all threads on all EUs in the system. This System Routine is defined per-context and is identified via a System IP (SIP) register in context state. At the time of each context switch, the appropriate SIP for that context is loaded into each EU, allowing each context to have custom implementation of exception handling routines if so desired.

The mechanism does not support handling recursive system routine access. This means when a thread cannot be asynchronously interrupted to an exception when executing a SIP.

Example:

An Exception is not supported when hardware is executing a SIP for context save and restore operations.

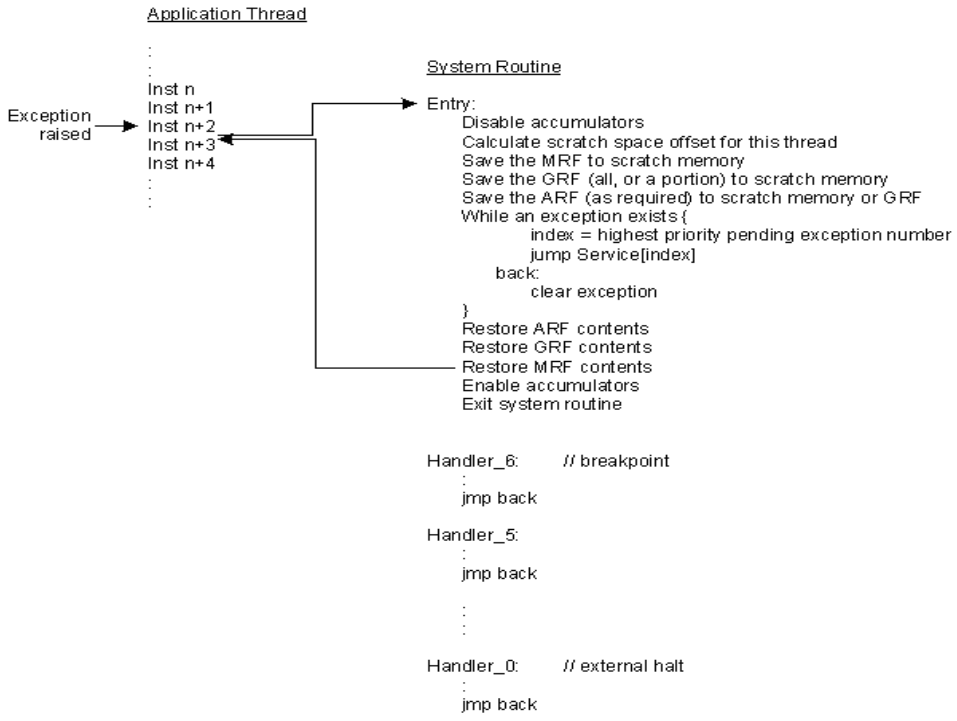
Exception-Related Architecture Registers

Exception-related registers are architecture registers cr0.0 through cr0.2. These registers are instantiated on a per-thread basis providing each hardware thread with unique control over exception recognition and handling. The registers provide the capability to mask exception types, determine the type of raised exception, store the return address, and control exiting from the System Routine back to the application thread.

Many of the bits in these registers are manipulated by both hardware and software. In all cases, the read/write operations by hardware and software occur at exclusive times in a thread's lifetime, thus there is no need for atomic read-modify-write operations when accessing these registers.

System Routine

The following diagram illustrates the basic flow of exception handling and the structure of the System Routine.



Invoking the System Routine

The System Routine is invoked in response to a raised exception.

Once an exception is raised, no further instructions from the application thread are issued until the System Routine has executed and returned control back to the application thread.

After an exception is recognized by hardware, the EU saves the thread's IP into its AIP register (cr0.2), and then moves the System Routine offset, SIP, into the thread's IP register. At this point the next instruction issued for that thread is the first instruction of the System Routine.

The System Routine maintains the same execution priority, GRF register space, and thread state as the application thread from which it is invoked.

Due to assuming the same priority, there may be significant absolute time between an exception being raised and invoking the System Routine, as other higher priority threads within the EU continue to execute. From a thread's perspective, once an exception is recognized, the next instruction issued is from the System Routine.

At the time of System Routine invocation, there may still be outstanding registers in-flight from the application thread. Depending on the instruction sequence in the System Routine, an in-flight register may be referenced by the System Routine and cause a register-in-flight dependency. These dependencies are honored by the System Routine and may cause the System Routine to be suspended until the register retires.

Exception processing is not nested within the System Routine. If a future exception is detected while executing the System Routine, the exception is latched into cr0.1, but does not cause a nested re-invocation of the System Routine. The exception recognition hardware recognizes only one outstanding exception of each type; i.e., once a specific exception type is detected and latched in cr0.1, and until the exception is cleared, any further exception of that type is lost.

Accumulators are not natively preserved across the System Routine. To make sure the accumulators are in the identical state once control is returned to the application thread, the System Routine must either set the Accumulator Disable bit of cr0.0 before using any instruction that modifies an accumulator or save and restore the accumulators (using GRF registers or system thread scratch memory) around the System Routine. Saving and restoring accumulators, including their extended precision bits, can be accomplished by a short series of moves and shifts of the accumulator register. Also note that the state of the Accumulator Disable bit itself must be preserved unless, by convention, the driver software limits its manipulation to only the System Routine.

Further, upon System Routine entry, the execution-related masks (Continue, Loop, If, and Active masks, contained in the Mask Register) will remain set as they were in the application thread. Thus, only a subset of channels may be active for execution. To enable execution on all channels, the System Routine may choose to use the instruction option 'NoMask', or may choose to set the mask registers to the desired value so long as it saves/restores the original masks upon System Routine entry/exit.

Similarly, there is no hardware mechanism to preserve flags, mask-stacks, or other architecture registers across the System Routine. The System Routine must ensure that these values are preserved (see the Conditional Instructions Within the System Routine section for related information).

ERRATA: Any instruction that is hanging due to an un-resolved dependency is non-interruptible, and will block execution of system routine. This includes sync.bar instruction waiting on a barrier clear, any $sbid dependency waiting on memory access, or sendc instruction waiting on other threads to clear the dependency.

Returning to the Application Thread

Prior to returning control to the application thread, the System Routine should clear the proper Exception Status and Control bit in cr0.1. Failure to do so forces the thread's execution to reenter the System Routine before any further instructions are executed from the application thread. (Note that single-stepping functionality is the one exception where the exception's Status and Control bit is not reset before exit.)

The System Routine may choose to loop within a single invocation of the System Routine until all pending exceptions are serviced or may choose to service exceptions one at a time (a simpler solution, but less efficient).

The System Routine is exited, and control returned to the application thread, via a write to the Primary Exception State and Control bit in cr0.0. Upon clearing this bit, the value of AIP (cr0.2) is restored to the thread's IP register and, with no further exceptions pending, execution resumes at that address. The System Routine must follow any write to the Primary Exception State and Control bit with at least one SIMD-16 *nop* instruction to allow control to transition. Throughout the System Routine, the AIP register maintains its value at the time the exception was raised unless directly modified by the System Routine. (See the AIP register definition for specifics on the IP value saved to AIP).

System IP (SIP)

The System IP (SIP) is the 16 byte-aligned offset of the first instruction of the System Routine, relative to the General State Base Address. SIP is assigned by the STATE_SIP command to the command streamer which updates SIP in the EU.

The SIP is widened to 48 bits. However, the EU still only uses the low 32 bits (bits 31:4 with bits 3:0 as zero).

When the System Routine is invoked, the application thread's current IP is first saved into the AIP field of cr0.2. The SIP address is then loaded into the thread's IP register and execution continues within the System Routine. Thus, each invocation of the System Routine has a common entry point. Returning from the System Routine loads IP from AIP, continuing thread execution.

System Routine Register Space

The System Routine uses the same GRF space as the thread that invokes it.

As such all of the calling thread's registers and their contents are visible to the System Routine. Further, the System Routine must only use r0..r15 of the GRF, as a minimal thread may have requested and been allocated this few. If the System Routine requires more registers than this, the driver should establish a higher minimum allocation for all threads.

The System Routine may encounter any residual register dependencies of the calling thread until such time that they clear by the return of in-flight writebacks.

Only one 32-bit GRF location, r0.4, is reserved for System Routine use. This location is sufficient to allow the System Routine to calculate the appropriate offset of its private scratch memory in the larger system scratch memory space (as dictated by binding table entry 254). The offset is left as a driver convention but is likely based on a combination of Thread and EU IDs (see the example system handler in the System Scratch Memory Space) section. Other than the reserved r0.4 register field, there is no explicit GRF register space dedicated to the System Routine, and any GRF needs must be accomplished via (a) convention between the System Routine and application code, or (b) the System Routine temporarily spilling the thread's GRF register contents to scratch memory and restoring those contents before System Routine exit.

No persistent storage is automatically allocated to the System Routine, although a driver implementation may set aside part of system scratch memory for the System Routine.

Any parameter passing to the System Routine (for use by software exceptions) is done via the GRF based on system thread/application thread convention.

System Scratch Memory Space

There is a single unified system scratch memory space per context shared by all EUs. It is anticipated that block is further partitioned into a unique scratch sub-space per thread via conventions implemented in the System Routine, with each hardware thread having a uniform block size at a calculated offset from the base address. The block address for a thread can be based on an offset derived from the thread's execution unit ID and thread ID made available through the TID and EUID field of architecture register sr0.0.

```
Per_Thread_Block_Size = System_Scratch_Block_Size / (EU_Count * Thread_Per_EU);

Offset = (sr0.0.EID * Threads_Per_EU + sr0.0.TID) * Per_Thread_Block_Size;
```

where in GFX:

Threads_Per_EU = 8

EU_Count = 8

System_Scratch_Block_Size is a driver choice.

Access to system scratch memory is performed through the Data Port via linear single register or block-based read/write messages. The driver may choose to use any binding table index for system scratch surface description. As a practical matter, the same index is expected to be used across all binding tables, as the index is typically hard coded in Data Port messages used within the System Routine coupled with the fact that a single System Routine is used for all threads. Read/write messages to the Data Port contain the address of the binding table (provided in r0 of all threads) and an offset, from which the Data Port calculates the final target address.

It is expected that the system scratch memory space is allocated by the driver at context-create time and remains persistent at a constant memory address throughout a context's lifetime.

Conditional Instructions Within the System Routine

It is expected that most, if not all, control flow within the System Routine is scalar in nature. If so, the System Routine should set SPF (Single Program Flow, cr0.0) to enable scalar branching. In this mode, conditional/loop instructions do not update the mask stacks and therefore do not have restrictions on their use nor require the save/restore of hardware mask stack registers.

If SIMD branching is desired within the System Routine, special considerations must be taken. Upon entry to the System Routine, the depth of the mask stacks is unknown at that point and may be near full. If so, a subsequent conditional instruction and its associated mask 'push' may cause a stack overflow. This would generate an exception within the system routine, an unsupported occurrence. To prevent this, if the System Routine uses SIMD conditional instructions, it must save the mask stacks prior to the first SIMD conditional instruction and restore them after the last SIMD conditional instruction. As a general solution, it may be easiest to simply implement the save/restore as part of the entry/exit code sequence, using an available GRF register pair as a storage location. Once saved, the stacks should be reset to their empty condition, namely depth = 0 and top of stack = 0xFFFFFFFF.

Exception Descriptions

This section describes conditions that can cause exceptions and transfer control to the System Routine.

Illegal Opcode

The ISA defines a single *illegal* opcode. The byte value of the *illegal* opcode is 0x00 due to it being a likely byte value encountered by a wayward instruction pointer value. The *illegal* instruction signals an exception if exception handling is enabled and invokes the system interrupt routine. If exception handling is NOT enabled, the illegal opcode is executed resulting in undetermined behavior including a system hang. Hardware decodes all legal opcodes supported. Any byte value that is not in the legal opcode list is decoded as an illegal opcode to trigger exception.

Undefined Opcodes

All undefined opcodes in the 8-bit opcode space (which includes instruction bit 7, reserved for future opcode expansion) are detected by hardware. If an undefined opcode is detected, the opcode is overridden by hardware, forcing the opcode value within the pipeline to the defined *illegal* opcode. The offending instruction, should it eventually be issued down the execution unit's pipeline, generates an Illegal Opcode exception as described in the section Illegal Opcode. The memory location of the offending opcode keeps its original value. That location can be queried to determine the opcode value.

Software Exception

A mechanism is provided to allow an application thread to invoke an exception and is triggered using the Software Exception Set and Clear bit of cr0.1. Sub-function determination and parameter passing into and out of the exception handler is left to convention between the system-thread and application-thread. The thread's IP is incremented before saving AIP and entering the System Routine, causing execution to resume at the next application-thread instruction after returning from the System Routine.

Context Save and Restore

The System Routine is also used to save and restore the context of the Execution Unit. This feature is enabled in GPGPU workloads *only*.

When the execution engine receives a preemption or an interrupt, the application thread invokes the System Routine. The System Routine is invoked only when all in-flight registers have retired. The system routine is used to save all the state of the EU to memory. When the sequence is complete, the primary exception control bit is cleared. This action stops all execution for the given thread and invalidates the thread. This means a new thread from a different context may be loaded. When the primary exception control bit is cleared, software must ensure that all outstanding messages from the EU are dispatched out of the execution message pipeline. This is achieved by creating a dependency on the last send that is saving EU state. A dummy instruction before clearing the primary exception control bit ensures that this is achieved.

The System Routine is also invoked on a context restore request. In this case a dummy thread is loaded into the EU which starts with the System Routine. This routine now restores the state of the EU. The restore sequence used in such a case should be consistent with the save sequence to ensure that state is restored correctly. After completing the restore sequence, the System Routine must clear the primary exception control bit in the Control Register. This enables hardware to switch to the application thread which continues execution.

Programming Note	
Context:	Context Save and Restore
When context save and restore is required to be supported for GPGPU workloads, Stack Overflow exception handling will not be supported. Software will either need to ensure stack is either completely disabled OR used in such a way, an exception will not trigger.	

Events That Do Not Generate Exceptions

The conditions described in this section are either not recognized or do not generate an exception.

Illegal Instruction Format

This condition includes malformed instructions in which the opcode is legal, but the source or destination operands or other instruction attributes do not comply with the instruction specification. There is no direct hardware support to detect these cases and the outcome of issuing a malformed instruction is undefined.

Malformed Message

A message's contents, destination registers, lengths, and descriptors are not interpreted in any way by the execution unit. Errors in specifying message fields do not raise exceptions in the EU but may be detected and reported by the shared functions.

GRF Register Out of Bounds

Unique GRF storage is allocated to each thread which, at a minimum, satisfies the register requirements specified in the thread's declaration. References to GRF register numbers beyond that called for in the thread's declaration do not generate exceptions. Depending on the implementation, out-of-bounds register numbers may be remapped to r0..r15, although this functionality should not be relied upon by the thread. The hardware guarantees the isolation of each thread's register space, thus there is no possibility of direct register manipulation via an out-of-bounds register access.

Hung Thread

There is no hardware mechanism in the EU to detect a hung thread and such a thread may remain hung indefinitely. It is expected that one or more hung threads will eventually cause the driver to recognize a context timeout and take appropriate recovery action.

Instruction Fetch Out of Bounds

The EU implements a full 32-bit instruction address range (with the 4 LSBs don't care), making it possible for a thread to attempt to jump to any 16-byte aligned offset in the 32-bit instruction address range. (Instruction addresses are offsets from the General State Base Address.) The EU does not provide any type of address checking on instruction fetch requests sent to the memory/cache hierarchy, although error conditions for memory addresses are reported via the Page Table Error Register and other memory interface registers.

FPU Math Errors

The EU's floating-point units (FPUs) have defined behaviors for traditional floating-point errors and do not generate exceptions. There is no support for signaling FPU math errors as exceptions.

Adds the IEEE Exception Trap Enable bit, which enables trapping IEEE exception flags. If enabled, IEEE floating-point exceptions set sticky bits in the IEEE Exceptions field of sr0.1. Note that IEEE floating-point exceptions still do *not* transfer control to any handler.

Computational Overflow

Depending on source operand types and values, destination type, and the operation being performed, overflows may occur in the execution pipelines. Many instructions support the overflow (**.o**) conditional modifier that assigns flag bits based on whether or not an overflow occurs.

The EU never signals exceptions for overflows. Software must provide any overflow handling.

System Routine Example

The following code sequence illustrates some concepts of the System Routine. It is intended to be just a shell, without getting into the specifics of each exception handler.

This example contains code for the message registers in the MRF. Other code in this example is useful for other processor generations.

The example frees enough MRF and GRF space to get the routine started, then jumps to the handler for the specific exception. Many other implementations are also valid, including single exception servicing (as opposed to looping) per invocation, and saving only the GRF or MRF space required by the exception being serviced.

```
#define ACC_DISABLE_MASK 0xFFFFFFFD
#define PRIMARY_EXCP_MASK 0x7FFFFFFF
#define SYSROUTINE_SCRATCH_BLKSIZE 16384 // for example

// Shared function IDs:
#define DPR 0x04000000
#define DPW 0x05000000

// Message lengths:
#define ML5 0x00500000
#define ML9 0x00900000

// Response lengths:
#define RL0 0x00000000
#define RL4 0x00040000
#define RL8 0x00080000

// Data port block sizes:
#define BS1_LOW  0x0000
#define BS1_HIGH 0x0100
#define BS2      0x0200
#define BS4      0x0300

// Scratch Layout:
#define SCR_OFFSET_MRF 0           //
#define SCR_OFFSET_GRF 512         // + 16 MRF registers
#define SCR_OFFSET_ARF 512 + 4096  // + 16 MRF + 128 GRF registers

// Write data port constants:
// target=dcache, type= oword_block_wr, binding_tbl_offset=0
#define DPW 0x000

// Read data port constants:
// target=dcache, type= oword_block_rd, binding_tbl_offset=0
#define DPR 0x000

Sys_Entry:  // Entry point to the System Routine.

// Disable accumulator for system routine:
and (1) cr0.0 cr0.0 ACC_DISABLE_MASK {NoMask}

// Calc scratch offset for this thread into r0.4:
shr (1) r0.4 sr0.0:uw 6 {NoMask}
add (1) r0.4 r0.4 sr0.0:ub {NoMask}
mul (1) r0.4 r0.4 SYSROUTINE_SCRATCH_BLKSIZE {NoMask}

// Setup m0 with block offset:
mov (8) m0 r0{NoMask}

// Save MRF 7..0 (may choose to save the whole MRF).
add (1) m0.2 r0.4 SCR_OFFSET_MRF {NoMask}
send (8) null m0 null DPW|ML9|RL0 {NoMask}

// Save MRF 8..15 (optional; req'ed if sys-routine stays w/in mrf7-0).
```

```
mov (8) m7 r0 {NoMask}
add (1) m7.2 r0.4 (SCR_OFFSET_MRF + 256) {NoMask}
send (8) null m7 null DPW|ML9|RL0 {NoMask}

// Save r0..r1 to system scratch:
// Note: done as a single register to guarantee external visibility
// See
mov (16) m1 r0 {NoMask}
send (8) m0 null null DPW|ML2|RL0 {NoMask}

// Save r2..r3 to free some room:
mov (16) m3 r2 {NoMask}
add (1) m0.2 r0.4 SCR_OFFSET_GRF + 64 {NoMask}
send (8) m0 null null DPW|ML4|RL0 {NoMask}

// Save r4..r7 to free some room (optional, depending on needs):
mov (16) m8 r4 {NoMask}
mov (16) m10 r6 {NoMask}
add (1) m7.2 r0.4 (SCR_OFFSET_GRF + 128) {NoMask}
send (8) m7 null null DPW|ML5|RL0 {NoMask}

// Save r8..r11 to free some room (optional, depending on needs):
mov (16) m1 r8 {NoMask}
mov (16) m3 r10 {NoMask}
add (1) m0.2 r0.4 (SCR_OFFSET_GRF + 256) {NoMask}
send (8) m0 null null DPW|ML5|RL0 {NoMask}

// Save r12..r15 to free some room (optional, depending on needs):
mov (16) m8 r12 {NoMask}
mov (16) m10 r14 {NoMask}
add (1) m7.2 r0.4 (SCR_OFFSET_GRF + 384) {NoMask}
send (8) m7 null null DPW|ML5|RL0 {NoMask}

// Save selected ARF registers (optional, depending on use):
// flags, others ...
// Save f0.0:
mov (1) r1.0:uw f0.0 {NoMask}

Next:  // Exceptions pending? If not, exit.

cmp.e (1) f0.0 cr0.4:uw 0:uw {NoMask}
(f0.0) mov (1) IP EXIT {NoMask}

// Find highest priority exception:
lzd (1) r1.1:uw cr0.4:uw {NoMask}

// Jump table to service routine:
jmpi (1) r1.1:uw{NoMask}
mov (1) IP CRService_0 {NoMask}
mov (1) IP CRService_1 {NoMask}
mov (1) IP CRService_2 {NoMask}
...
mov (1) IP CRService_15{NoMask}
mov (1) IP Next

Service_0:
// Clear exception from cr0.1.
// Perform service routine.
// Jump to exit (or if looping on exceptions, go to next loop).
...
Service_15:
// Clear exception from cr0.1.
// Perform service routine.
// Jump to exit (or if looping on exceptions, go to next loop).
```

```
Exit:
// Restore f0.0.
// Restore other ARF registers (as required).
// Restore r12..r15.
// Restore r8..r11.
// Restore r4..r7.
// Restore r0..r3.
// Restore m8..m15.
// Restore m0..m7.
// Clear Primary Exception State bit in cr0.0:
and (1) cr0.0 cr0.0 PRIMARY_EXCP_MASK
nop (16)
```

Below is a code sequence to programmatically clear the GRF scoreboard in case of a timeout waiting on a register that may never return.

At this point, all we know is we have a hung thread. We'd like to copy the GRF to scratch memory to make it visible, but there may be a register that is hung with an outstanding dependency. To get around any hung dependency, walk the GRF using NoDDChk, using an execution mask of f0 == 0 so we don't touch the register contents.

```
Clear_Dep:
mov f0 0x00
(f0) mov r0 0x00 {NoDDChk}
(f0) mov r1 0x00 {NoDDChk}
(f0) mov r2 0x00 {NoDDChk}
...
(f0) mov r127 0x00 {NoDDChk}
// GRF scoreboard now cleared.
```

Instruction Set Summary

Instruction Set Characteristics

SIMD Instructions and SIMD Width

Instructions are SIMD (single instruction multiple data) instructions. The number of data elements per instruction, or the execution size, depends on the data type. For example, the execution size for instructions operating on 256-bit wide vectors can be up to 8 for 32-bit data types, and be up to 16 for 16-bit data. The maximum execution size for instructions for 8-bit data types is also limited to 16.

An instruction compression mode is supported for 32-bit instructions (including mixed 32-bit and 16-bit data computation). A compressed instruction works on twice as much SIMD data as that for a non-compressed instruction. A compressed instruction is converted into two native instructions by the instruction dispatcher in the EU.

Instructions are executed on a narrower SIMD execution pipeline. Therefore, native instructions take multiple execution cycles to complete.

Instruction Operands and Register Regions

Most instructions may have up to three operands, two sources and one destination. Each operand is able to address a register region. Source operands support negate and absolute modifier and channel swizzle, and the destination operand supports channel mask.

Dual destination instructions are also supported (four-operand instructions in a general sense): One case is for the implied destination - flag register, where the conditional modifiers and the predicate modifiers may apply. Another case is the message header creation (implied move or implied assembling of the header) in the *send* instruction.

Each execution channel contains an accumulator that is wider than the input data to support back-to-back accumulation operations with increased precision. The added precision (see accumulator register description in Execution Environment chapter) determines the maximum number of accumulations before possible overflow. The accumulator can be pre-loaded through the use of *mov*. It can also be pre-loaded by arithmetic instructions such as *add* or *mul*, since the result of these instructions can go to the accumulator. The accumulator registers are per thread and therefore safe for thread switching.

Register access can be direct or register indirect. Register-indirect register access uses address registers plus an immediate offset term to compute the register addresses, and only applies to the first source operand (src0) and/or the destination operand.

There is one address register.

There are 16 address sub-registers.

Each sub-register contains a 16-bit unsigned value. The leading two sub-registers form a special doubleword that can be used as the descriptor for the send instruction.

Source operand can also be immediate value (also referred to as inline constants). For instructions with two source operands, only the second operand src1 is allowed to be immediate. For instructions with only one source operand, the source operand src0 is used and it can be immediate.

An immediate source operand can be a scalar value of specified type up to 32-bit wide, which is replicated to create a vector with length of Execution Size. An immediate operand can also be a special 32-bit vector with 8 elements each of 4-bit signed integer value, or a 32-bit vector with 4 elements each of 8-bit restricted float value.

Instruction Execution

It is implied that all instructions operate across all channels of data unless otherwise specified either via destination mask, predication, execution mask (caused by SIMD branch and loop instructions), or execution size.

Instruction execution size can be specified per instruction, from scalar (*ExecSize* = 1) up to the maximal execution size supported for the data type, with the restriction that execution size can only be in power of 2.

EU Compact Instructions

On receiving an instruction with bit 29 (CmptCtrl) set, HW recognizes it as a 64-bit compact instruction. Hardware then uses the index fields inside the compact instruction to lookup values in the associated compaction tables, then uses the table outputs along with other fields in the compact instruction to reconstruct the 128-bit native-sized instruction.

All flow control instructions use the new offset format, a signed 32-bit offset in units of bytes.

The native 128-bit instruction format provides access to all instruction options. Only some instruction options and combinations of instruction options can be represented in the compact instruction formats.

Which native instructions can be represented as compact instructions and the details of the compact instruction formats, and the compaction tables used may change with each processor generation.

In the following instruction format tables, the Mapping Bits and Mapping Description columns describe the mappings into native instruction fields.

EU Compact Instruction Format

This page contains an outline of the compacted (64bit) encodings of various instructions. For the meaning of various fields (e.g. allowed values) consult with the Instruction Fields and EU Instruction Compaction Tables page. Branch Instruction and Send instructions are not compacted.

Compact Instructions Format

<table>
<tr><th>Bit</th><th>2src</th><th>2src-imm</th><th>3src</th></tr>
<tr><td>63</td><td rowspan="7">Src1.Reg</td><td rowspan="11">ImmValue2[11:0]</td><td rowspan="7">Src1.Reg</td></tr>
<tr><td>62</td></tr>
<tr><td>61</td></tr>
<tr><td>60</td></tr>
<tr><td>59</td></tr>
<tr><td>58</td></tr>
<tr><td>57</td></tr>
<tr><td>56</td><td></td><td></td><td></td></tr>
<tr><td>55</td><td rowspan="4">Src1.Index2</td><td rowspan="10">Src2.Reg</td></tr>
<tr><td>54</td></tr>
<tr><td>53</td></tr>
<tr><td>52</td></tr>
<tr><td>51</td><td rowspan="4">Src0.Index2</td><td rowspan="4">Src0.Index2</td></tr>
<tr><td>50</td></tr>
<tr><td>49</td></tr>
<tr><td>48</td></tr>
</table>

Bit	2src	2src-imm	3src
47	Src0.Reg	Src0.Reg	Src0.Reg
46			
45			
44			
43			
42			
41			
40			
39	SubRegIndex2	SubRegIndex2	SubRegIndex3
38			
37			
36			
35			
34	DataTypeIndex2	DataTypeIndex2	SourceIndex3
33			
32			
31			
30			
29	CmptCtrl = 1	CmptCtrl = 1	CmptCtrl = 1
28	ControlIndex2	ControlIndex2	ControlIndex3
27			
26			
25			
24			
23	Dst.Reg	Dst.Reg	Dst.Reg
22			
21			
20			
19			
18			
17			
16			
15	SWSB	SWSB	SWSB
14			

Bit	2src	2src-imm	3src
13			
12			
11			
10			
9			
8			
7	DbgCtrl	DbgCtrl	DbgCtrl
6	**Opcode** =(unary/binary/ math/sync op)	**Opcode** =(unary/binary/ math/sync op)	**Opcode** =(ternary op)
5			
4			
3			
2			
1			
0			

EU Instruction Compaction Tables

Unary/Binary Instruction Compaction Tables

The following five tables describe the mappings for the ControlIndex2, DataTypeIndex2, SubRegIndex2, Src0Index2, and Src1Index2 fields in the Unary/Binary Compact Instruction format. The values below are separated with commas and spaces for readability but represent a binary number MSB to LSB.

ControlIndex2 Compact Instruction Field Mappings

ControlIndex2	21-Bit Mapping { CondCtrl[3:0],Saturate,AccWrCtrl,AtomicCtrl,MaskCtrl, PredInv,PredCtrl[3:0], FlagRegNum,FlagSubRegNum, ChOff[2:0],ExecSize[2:0] }	Mapped Meaning
0	0000,0,0,0,0, 0,0000, 0,0, 000,100	(16|M0)
1	0000,0,0,0,0, 0,0000, 0,0, 000,011	(8|M0)
2	0000,0,0,0,1, 0,0000, 0,0, 000,000	(W) (1|M0)
3	0000,0,0,0,1, 0,0000, 0,0, 000,100	(W) (16|M0)
4	0000,0,0,0,1, 0,0000, 0,0, 000,011	(W) (8|M0)
5	0100,0,0,0,0, 0,0000, 0,0, 000,100	(16|M0) (ge)f0.0
6	0000,0,0,0,0, 0,0000, 0,0, 100,100	(16|M16)
7	0101,0,0,0,0, 0,0000, 0,0, 000,100	(16|M0) (lt)f0.0
8	0000,0,0,0,0, 0,0000, 0,0, 000,000	(1|M0)

ControlIndex2	21-Bit Mapping { CondCtrl[3:0],Saturate,AccWrCtrl,AtomicCtrl,MaskCtrl, PredInv,PredCtrl[3:0], FlagRegNum,FlagSubRegNum, ChOff[2:0],ExecSize[2:0] }	Mapped Meaning
9	0000,1,0,0,0, 0,0000, 0,0, 000,100	(16|M0) (sat)
10	0000,0,0,0,0, 0,0000, 0,0, 010,011	(8|M8)
11	0011,0,0,0,0, 0,0000, 0,0, 000,100	(16|M0) (gt)f0.0
12	0001,0,0,0,0, 0,0000, 0,0, 000,100	(16|M0) (eq)f0.0
13	0001,0,0,0,1, 0,0000, 0,0, 000,100	(W) (16|M0) (eq)f0.0
14	0010,0,0,0,0, 0,0000, 0,0, 000,100	(16|M0) (ne)f0.0
15	0000,0,0,0,0, 0,0001, 0,0, 000,100	(f0.0) (16|M0)
16	0101,0,0,0,0, 0,0000, 0,0, 000,011	(8|M0) (lt)f0.0
17	0000,0,0,0,0, 0,0001, 1,0, 000,100	(f1.0) (16|M0)
18	0000,0,0,0,1, 0,0000, 0,0, 000,001	(W) (2|M0)
19	0000,0,0,0,0, 0,0001, 0,1, 000,100	(f0.1) (16|M0)
20	0000,0,0,0,0, 0,0001, 1,1, 000,100	(f1.1) (16|M0)
21	0100,0,0,0,1, 0,0000, 0,0, 000,100	(W) (16|M0) (ge)f0.0
22	0000,0,0,0,0, 0,0000, 0,0, 100,011	(8|M16)
23	0000,0,0,0,0, 0,0000, 0,0, 110,011	(8|M24)
24	0101,0,0,0,1, 0,0000, 0,0, 000,100	(W) (16|M0) (lt)f0.0
25	0100,0,0,0,0, 0,0000, 0,0, 000,011	(8|M0) (ge)f0.0
26	0001,0,0,0,1, 0,0000, 0,0, 000,000	(W) (1|M0) (eq)f0.0
27	0000,1,0,0,0, 0,0000, 0,0, 000,011	(8|M0) (sat)
28	0101,0,0,0,0, 0,0000, 1,0, 000,100	(16|M0) (lt)f1.0
29	0001,0,0,0,0, 0,0000, 0,0, 000,011	(8|M0) (eq)f0.0
30	0000,0,1,0,0, 0,0000, 0,0, 000,011	(8|M0) {AccWrEn}
31	0000,0,0,0,1, 0,0000, 0,0, 100,100	(W) (16|M16)

DataTypeIndex2 Compact Instruction Field Mappings

DataTypeIndex2	20-Bit Mapping { Src1.RegFile[0], Src1.SrcType[3:0], Src0.RegFile[0], Dst.RegFile[0], Dst.HorzStride[1:0], Src1.IsImm, Src0.IsImm, Src0.SrcType[3:0], Dst.DstType[3:0], Dst.AddrMode }	Mapped Meaning { dst, src0, src1 }
0	1,1010,1,1,01, 0,0, 1010,1010,0	grf<1>:f grf:f grf:f
1	0,0000,1,1,01, 0,0, 1010,1010,0	grf<1>:f grf:f arf:ub
2	0,0000,0,1,01, 0,1, 1010,1010,0	grf<1>:f imm:f
3	0,1010,1,1,01, 1,0, 1010,1010,0	grf<1>:f grf:f imm:f
4	1,1010,1,0,01, 0,0, 1010,1010,0	arf<1>:f grf:f grf:f
5	1,1010,0,1,01, 0,0, 1010,1010,0	grf<1>:f arf:f grf:f
6	0,1010,1,0,01, 1,0, 1010,1010,0	arf<1>:f grf:f imm:f
7	0,0000,0,0,01, 0,0, 0000,0000,0	arf<1>:ub arf:ub arf:ub
8	1,1010,0,0,01, 0,0, 1010,1010,0	arf<1>:f arf:f grf:f
9	0,0101,1,1,01, 1,0, 0110,0110,0	grf<1>:d grf:d imm:w
10	1,0110,1,1,01, 0,0, 0110,0110,0	grf<1>:d grf:d grf:d
11	0,1010,0,1,01, 1,0, 1010,1010,0	grf<1>:f arf:f imm:f
12	1,0010,1,1,01, 0,0, 0010,0010,0	grf<1>:ud grf:ud grf:ud
13	0,1010,0,0,01, 1,0, 1010,1010,0	arf<1>:f arf:f imm:f
14	0,0110,1,1,01, 1,0, 0110,0110,0	grf<1>:d grf:d imm:d
15	0,0010,1,1,01, 1,0, 0010,0010,0	grf<1>:ud grf:ud imm:ud
16	0,0000,1,1,10, 0,0, 1010,1010,0	grf<2>:f grf:f arf:ub
17	0,0101,1,0,01, 1,0, 0110,0110,0	arf<1>:d grf:d imm:w
18	0,0000,0,0,01, 0,0, 0001,0001,0	arf<1>:uw arf:uw arf:ub

DataTypeIndex2	20-Bit Mapping { Src1.RegFile[0], Src1.SrcType[3:0], Src0.RegFile[0], Dst.RegFile[0], Dst.HorzStride[1:0], Src1.IsImm, Src0.IsImm, Src0.SrcType[3:0], Dst.DstType[3:0], Dst.AddrMode }	Mapped Meaning { dst, src0, src1 }
19	0,0000,0,1,01, 0,0, 0010,0010,0	grf<1>:ud arf:ud arf:ub
20	0,0100,1,1,01, 1,0, 0001,0101,0	grf<1>:w grf:uw imm:v
21	0,0001,1,1,01, 1,0, 0001,0001,0	grf<1>:uw grf:uw imm:uw
22	1,0010,1,1,10, 0,0, 0010,0010,0	grf<2>:ud grf:ud grf:ud
23	0,0000,1,1,01, 0,0, 1010,0110,0	grf<1>:d grf:f arf:ub
24	1,0001,1,0,01, 0,0, 0110,0110,0	arf<1>:d grf:d grf:uw
25	0,0000,1,1,01, 0,0, 0010,1010,0	grf<1>:f grf:ud arf:ub
26	0,0101,1,1,01, 1,0, 0010,0110,0	grf<1>:d grf:ud imm:w
27	0,0000,0,1,01, 0,0, 0001,0001,0	grf<1>:uw arf:uw arf:ub
28	0,0000,1,1,01, 0,0, 0001,1010,0	grf<1>:f grf:uw arf:ub
29	0,0000,1,1,01, 0,0, 0000,1010,0	grf<1>:f grf:ub arf:ub
30	0,0000,1,1,01, 0,0, 0110,1010,0	grf<1>:f grf:d arf:ub
31	0,0000,0,1,01, 0,0, 1010,1010,0	grf<1>:f arf:f arf:ub

SubRegIndex2 Compact Instruction Field Mappings

SubRegIndex2	15-Bit Mapping { Src1.SubRegNum[4:0], Src0.SubRegNum[4:0], Dst.SubRegNum[4:0] }	Mapped Meaning { dst, src0, src1 }
0	00000,00000,00000	.0 .0 .0
1	10000,00000,00000	.0 .0 .16
2	00100,00000,00000	.0 .0 .4
3	01100,00000,00000	.0 .0 .12
4	00000,00100,00000	.0 .4 .0
5	01000,00000,00000	.0 .0 .8
6	10100,00000,00000	.0 .0 .20

SubRegIndex2	15-Bit Mapping { Src1.SubRegNum[4:0], Src0.SubRegNum[4:0], Dst.SubRegNum[4:0] }	Mapped Meaning { dst, src0, src1 }
7	00000,00000,01000	.8 .0 .0
8	00000,01000,00000	.0 .8 .0
9	11000,00000,00000	.0 .0 .24
10	11100,00000,00000	.0 .0 .28
11	00000,10000,00000	.0 .16 .0
12	00000,00000,00100	.4 .0 .0
13	00000,11000,00000	.0 .24 .0
14	00000,10100,00000	.0 .20 .0
15	00000,01100,00000	.0 .12 .0
16	00000,11100,00000	.0 .28 .0
17	00000,00000,11100	.28 .0 .0
18	00000,00000,10000	.16 .0 .0
19	00000,00000,01100	.12 .0 .0
20	00000,00000,11000	.24 .0 .0
21	00000,00000,10100	.20 .0 .0
22	00000,00000,00010	.2 .0 .0
23	00000,01010,00000	.0 .10 .0
24	00000,00010,00000	.0 .2 .0
25	00000,00100,00100	.4 .4 .0
26	00000,00010,11100	.28 .2 .0
27	00000,00010,00010	.2 .2 .0
28	00000,01100,01100	.12 .12 .0
29	00000,00001,00000	.0 .1 .0
30	00000,00011,00000	.0 .3 .0
31	11000,11000,00000	.0 .24 .24

Src0Index2	12-Bit Mapping { Src0.VertStride[3:0], Src0.Width[2:0], Src0.AddrMode, Src0.HorzStride[1:0], Src0.Mod[1:0] }	Mapped Meaning
0	0001,000,0,00,00	r<1;1,0>
1	0000,000,0,00,00	r<0;1,0>
2	0001,000,0,00,10	-r<1;1,0>
3	0001,000,0,00,01	(abs)r<1;1,0>
4	0000,000,0,00,10	-r<0;1,0>
5	0010,000,0,00,00	r<2;1,0>

Src0Index2	12-Bit Mapping { Src0.VertStride[3:0], Src0.Width[2:0], Src0.AddrMode, Src0.HorzStride[1:0], Src0.Mod[1:0] }	Mapped Meaning
6	0010,010,0,00,00	r<2;4,0>
7	0011,010,0,00,00	r<4;4,0>
8	0011,000,0,00,00	r<4;1,0>
9	0001,000,0,00,11	-(abs)r<1;1,0>
10	0000,000,0,00,01	(abs)r<0;1,0>
11	1111,000,1,00,00	r[a]<1,0>
12	0100,011,0,00,00	r<8;8,0>
13	0001,010,0,00,00	r<1;4,0>
14	0100,010,0,10,00	r<8;4,2>
15	0010,000,0,00,10	-r<2;1,0>

Src1Index2	12-Bit Mapping { Src1.Mod[1:0], Src1.VertStride[3:0], Src1.Width[2:0], Src1.AddrMode, Src1.HorzStride[1:0] }	Mapped Meaning
0	00,0001,000,0,00	r<1;1,0>
1	00,0000,000,0,00	r<0;1,0>
2	10,0001,000,0,00	-r<1;1,0>
3	10,0000,000,0,00	-r<0;1,0>
4	01,0001,000,0,00	(abs)r<1;1,0>
5	10,0011,010,0,00	-r<4;4,0>
6	00,0010,000,0,00	r<2;1,0>
7	00,0011,010,0,00	r<4;4,0>
8	00,0011,000,0,00	r<4;1,0>
9	11,0001,000,0,00	-(abs)r<1;1,0>
10	01,0000,000,0,00	(abs)r<0;1,0>
11	11,0000,000,0,00	-(abs)r<0;1,0>
12	00,0100,011,0,00	r<8;8,0>
13	10,0010,000,0,00	-r<2;1,0>
14	10,0000,001,0,01	-r<0;2,1>
15	10,0001,000,1,00	-r[a]<1;1,0>

ImmValue2 Compact Instruction Field

Immediates values of up to 12 bits can be represented in ImmValue2 field of compacted format.

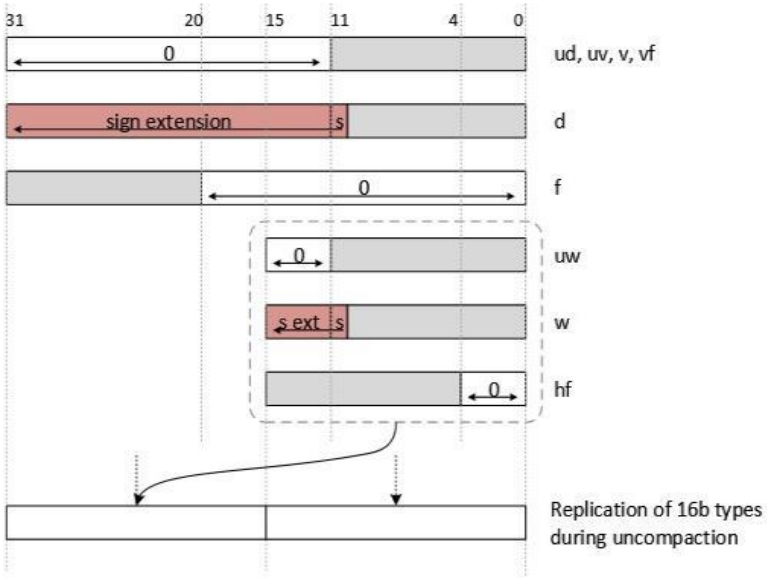
For unsigned integer types if bits [31:12] ([15:12] for 16b type) are all zero's, then bits [11:0] can be compacted as immediate value. Packed datatypes uv, v, vf are treated as 32 unsigned integer in the

context of compaction. During uncompaction zero is extended up to the MSB of datatype. 16bit datatypes are replicated to occupy 32bit of uncompacted immediate value.

For signed integer types if bits [31:11] ([15:12] for 16b type) are all zero's or all one's, then bits [11:0] can be compacted as immediate value. During uncompaction sign bit 11 is extended up to the MSB of datatype.16bit datatypes are replicated to occupy 32bit of uncompacted immediate value.

For float types if bits [19:0] ([3:0] for 16b type) are all zero's, then bits [31:20] can be compacted as immediate value. During uncompaction bits [19:0] ([3:0] for 16d type) are set to zero. 16bit datatypes are replicated to occupy 32bit of uncompacted immediate value.

64bit datatypes are not compacted.



Ternary Instruction Compaction Tables

The following three tables describe the mappings for the ControlIndex3, SourceIndex3 and SubRegIndex3 fields in the Ternary Compact Instruction format.

The following three tables describe the mappings for the ControlIndex3, SourceIndex3 and SubRegIndex3 fields in the Ternary Compact Instruction format.

ControlIndex3	37-Bit Mapping { CondMod[3:0], Src1.SrcType[2:0], Src2.SrcType[2:0], Dst.RegFile[0], SystolicDepth[1:0], Src0.SrcType[2:0], ExecDataType, Dst.DstType[2:0], Saturate, AccWrCtrl, AtomicCtrl, MaskCtrl, PredInv, PredCtrl[3:0], FlagRegNum, FlagSubRegNum, ChOff[2:0], ExecSize[2:0] }	Mapped Meaning
0	0000,010,010,1,00,010,1,010,0,0,0,0,0,0000,0,0,000,100	(16|M0) grf<1>:f :f :f :f
1	0000,010,010,1,00,010,1,010, 0,0,0,0,0,0000,0,0,000,011	(8|M0) grf<1>:f :f :f :f
2	0000,010,010,0,00,010,1,010, 0,0,0,0,0,0000,0,0,000,011	(8|M0) arf<1>:f :f :f :f
3	0000,010,010,1,00,010,1,010, 0,0,0,1,0,0000,0,0,000,011	(W) (8|M0) grf<1>:f :f :f :f
4	0000,010,010,0,00,010,1,010, 0,0,0,1,0,0000,0,0,000,011	(W) (8|M0) arf<1>:f :f :f :f
5	0000,010,010,0,00,010,1,010, 0,0,0,0,0,0000,0,0,010,011	(8|M8) arf<1>:f :f :f :f
6	0000,010,010,1,00,010,1,010, 0,0,0,0,0,0000,0,0,010,011	(8|M8) grf<1>:f :f :f :f
7	0000,010,010,0,00,010,1,010, 0,0,0,1,0,0000,0,0,010,011	(W) (8|M8) arf<1>:f :f :f :f
8	0000,010,010,1,00,010,1,010, 0,0,0,1,0,0000,0,0,010,011	(W) (8|M8) grf<1>:f :f :f :f
9	0000,010,010,1,00,010,1,010, 0,0,0,1,0,0000,0,0,000,100	(W) (16|M0) grf<1>:f :f :f :f
10	0000,010,010,0,00,010,1,010, 0,0,0,0,0,0000,0,0,000,100	(16|M0) arf<1>:f :f :f :f
11	0000,010,010,1,00,010,1,010, 1,0,0,0,0,0000,0,0,000,100	(16|M0) (sat)grf<1>:f :f :f :f
12	0000,010,010,1,00,010,1,010, 0,0,0,0,0,0000,0,0,100,100	(16|M16) grf<1>:f :f :f :f
13	0000,010,010,0,00,010,1,010, 0,0,0,1,0,0000,0,0,000,100	(W) (16|M0) arf<1>:f :f :f :f
14	0000,010,010,1,00,010,1,010, 0,0,0,1,0,0000,0,0,000,000	(W) (1|M0) grf<1>:f :f :f :f
15	0000,010,010,1,00,010,1,010, 1,0,0,0,0,0000,0,0,000,011	(8|M0) (sat)grf<1>:f :f :f :f
16	0000,010,010,0,00,010,1,010, 0,0,0,1,0,0000,0,0,100,011	(W) (8|M16) arf<1>:f :f :f :f
17	0000,010,010,0,00,010,1,010, 0,0,0,1,0,0000,0,0,110,011	(W) (8|M24) arf<1>:f :f :f :f
18	0000,010,010,1,00,010,1,010, 0,0,0,1,0,0000,0,0,100,011	(W) (8|M16) grf<1>:f :f :f :f
19	0000,010,010,1,00,010,1,010, 0,0,0,1,0,0000,0,0,110,011	(W) (8|M24) grf<1>:f :f :f :f
20	0000,010,010,0,00,010,1,010, 0,0,0,0,0,0000,0,0,110,011	(8|M24) arf<1>:f :f :f :f
21	0000,010,010,0,00,010,1,010, 0,0,0,0,0,0000,0,0,100,011	(8|M16) arf<1>:f :f :f :f
22	0000,000,100,1,11,110,0,110, 0,0,0,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :ub :b
23	0000,000,000,1,11,110,0,110, 0,0,1,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :ub :ub {Atomic}

ControlIndex3	37-Bit Mapping { CondMod[3:0], Src1.SrcType[2:0], Src2.SrcType[2:0], Dst.RegFile[0], SystolicDepth[1:0], Src0.SrcType[2:0], ExecDataType, Dst.DstType[2:0], Saturate, AccWrCtrl, AtomicCtrl, MaskCtrl, PredInv, PredCtrl[3:0], FlagRegNum, FlagSubRegNum, ChOff[2:0], ExecSize[2:0] }	Mapped Meaning
24	0000,100,100,1,11,110,0,110, 0,0,1,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :b :b {Atomic}
25	0000,100,000,1,11,110,0,110, 0,0,1,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :b :ub {Atomic}
26	0000,100,100,1,11,110,0,110, 0,0,0,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :b :b
27	0000,000,000,1,11,110,0,110, 0,0,0,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :ub :ub
28	0000,000,100,1,11,110,0,110, 0,0,1,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :ub :b {Atomic}
29	0000,100,000,1,11,110,0,110, 0,0,0,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:d :d :b :ub
30	0000,101,101,1,11,010,1,010, 0,0,1,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:f :f :bf :bf {Atomic}
31	0000,101,101,1,11,010,1,010, 0,0,0,0,0,0000,0,0,000,011	dpas.8x* (8|M0) grf<1>:f :f :bf :bf

SourceIndex3	21-Bit Mapping { Src2.RegFile[0], Src2.HorzStride[1:0], Src1.RegFile[0], Src1.HorzStride[1:0], Src1.VertStride[1], Src1.Mod[1:0], Src2.Mod[1:0], Src1.VertStride[0], Src0.RegFile[0], Src0.HorzStride[1:0], Src2.IsImm, Src0.IsImm, Src0.Mod[1:0], Src0.VertStride[1], Src0.VertStride[0] }	Mapped Meaning { src0, src1, src2 }
0	1,00,1,00,0,00,00, 1,1,00,0,0,00,0,0	grf<0;0> grf<1;0> grf<0>
1	1,00,1,00,0,00,00, 1,0,00,0,0,00,0,1	arf<1;0> grf<1;0> grf<0>
2	1,01,1,00,0,00,00, 1,1,00,0,0,00,0,1	grf<1;0> grf<1;0> grf<1>
3	1,00,1,00,0,00,00, 1,1,00,0,0,00,0,1	grf<1;0> grf<1;0> grf<0>
4	1,01,1,00,0,00,00, 0,1,00,0,0,00,0,1	grf<1;0> grf<0;0> grf<1>
5	1,01,1,00,0,00,00, 1,1,00,0,0,10,0,1	-grf<1;0> grf<1;0> grf<1>
6	1,01,0,00,0,00,00, 1,1,00,0,0,00,0,1	grf<1;0> arf<1;0> grf<1>
7	1,01,1,00,0,00,00, 1,1,00,0,0,00,0,0	grf<0;0> grf<1;0>

SourceIndex3	21-Bit Mapping { Src2.RegFile[0], Src2.HorzStride[1:0], Src1.RegFile[0], Src1.HorzStride[1:0], Src1.VertStride[1], Src1.Mod[1:0], Src2.Mod[1:0], Src1.VertStride[0], Src0.RegFile[0], Src0.HorzStride[1:0], Src2.IsImm, Src0.IsImm, Src0.Mod[1:0], Src0.VertStride[1], Src0.VertStride[0] }	Mapped Meaning { src0, src1, src2 }
		grf<1>
8	1,00,0,00,0,00,00, 1,1,00,0,0,00,0,0	grf<0;0> arf<1;0> grf<0>
9	1,01,1,00,0,00,10, 1,1,00,0,0,00,0,1	grf<1;0> grf<1;0> -grf<1>
10	1,01,1,00,0,10,00, 1,1,00,0,0,00,0,1	grf<1;0> -grf<1;0> grf<1>
11	1,01,1,00,0,00,00, 0,1,00,0,0,00,0,0	grf<0;0> grf<0;0> grf<1>
12	1,00,0,00,0,00,00, 1,1,00,0,0,00,0,1	grf<1;0> arf<1;0> grf<0>
13	1,00,1,00,0,10,00, 1,1,00,0,0,00,0,0	grf<0;0> -grf<1;0> grf<0>
14	1,00,1,00,0,10,00, 1,1,00,0,0,00,0,1	grf<1;0> -grf<1;0> grf<0>
15	1,00,1,00,0,00,00, 1,1,00,0,0,10,0,1	-grf<1;0> grf<1;0> grf<0>
16	1,00,1,00,0,00,00, 0,1,00,0,0,00,0,1	grf<1;0> grf<0;0> grf<0>
17	1,00,1,00,0,00,00, 1,1,00,0,0,10,0,0	-grf<0;0> grf<1;0> grf<0>
18	1,00,1,00,0,00,00, 0,1,00,0,0,00,0,0	grf<0;0> grf<0;0> grf<0> dpas.*x1 grf:d grf:[ub,b] grf:[ub,b] dpas.*x1 grf:f grf:bf grf:bf
19	1,01,1,00,0,10,00, 1,1,00,0,0,00,0,0	grf<0;0> -grf<1;0> grf<1>
20	1,00,1,00,0,00,10, 1,1,00,0,0,00,0,0	grf<0;0> grf<1;0> -grf<0>
21	1,01,0,00,0,00,00, 1,1,00,0,0,00,0,0	grf<0;0> arf<1;0> grf<1>
22	1,00,1,00,0,00,10, 1,1,00,0,0,00,0,1	grf<1;0> grf<1;0> -grf<0>
23	1,01,1,00,0,00,10, 1,1,00,0,0,10,0,1	-grf<1;0> grf<1;0> -grf<1>
24	1,00,1,00,0,10,00, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[u2,s2] grf:[ub,b]
25	1,00,1,00,0,00,10, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[ub,b] grf:[u2,s2]

SourceIndex3	21-Bit Mapping { Src2.RegFile[0], Src2.HorzStride[1:0], Src1.RegFile[0], Src1.HorzStride[1:0], Src1.VertStride[1], Src1.Mod[1:0], Src2.Mod[1:0], Src1.VertStride[0], Src0.RegFile[0], Src0.HorzStride[1:0], Src2.IsImm, Src0.IsImm, Src0.Mod[1:0], Src0.VertStride[1], Src0.VertStride[0] }	Mapped Meaning { src0, src1, src2 }
26	1,00,1,00,0,10,10, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[u2,s2] grf:[u2,s2]
27	1,00,1,00,0,01,00, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[u4,s4] grf:[ub,b]
28	1,00,1,00,0,01,10, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[u4,s4] grf:[u2,s2]
29	1,00,1,00,0,00,01, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[ub,b] grf:[u4,s4]
30	1,00,1,00,0,01,01, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[u4,s4] grf:[u4,s4]
31	1,00,1,00,0,10,01, 0,1,00,0,0,00,0,0	dpas.*x1 grf:d grf:[u2,s2] grf:[u4,s4]

SubRegIndex3 Compact Instruction Field Mappings

SubRegIndex3	20-Bit Mapping { Src2.SubRegNum[4:0], Src1.SubRegNum[4:0], Src0.SubRegNum[4:0], Dst.SubRegNum[4:0] }	Mapped Meaning { dst, src0, src1, src2 }
0	00000,00000,00000,00000	.0 .0 .0 .0
1	00100,00000,00000,00000	.0 .0 .0 .4
2	00000,00000,01100,00000	.0 .12 .0 .0
3	10100,00000,00000,00000	.0 .0 .0 .20
4	10000,00000,11100,00000	.0 .28 .0 .16
5	01100,00000,00000,00000	.0 .0 .0 .12
6	01000,00000,00000,00000	.0 .0 .0 .8
7	00000,01000,00000,00000	.0 .0 .8 .0
8	00000,00100,00000,00000	.0 .0 .4 .0
9	11000,00000,00000,00000	.0 .0 .0 .24
10	10000,00000,00000,00000	.0 .0 .0 .16
11	11100,00000,00000,00000	.0 .0 .0 .28
12	00000,11000,00000,00000	.0 .0 .24 .0
13	00000,00000,00100,00000	.0 .4 .0 .0
14	00000,10000,00000,00000	.0 .0 .16 .0
15	00000,01100,00000,00000	.0 .0 .12 .0
16	00000,10100,00000,00000	.0 .0 .20 .0
17	00000,11100,00000,00000	.0 .0 .28 .0
18	00000,00000,01000,00000	.0 .8 .0 .0
19	00000,00000,10000,00000	.0 .16 .0 .0
20	00000,00000,11000,00000	.0 .24 .0 .0

SubRegIndex3	20-Bit Mapping { Src2.SubRegNum[4:0], Src1.SubRegNum[4:0], Src0.SubRegNum[4:0], Dst.SubRegNum[4:0] }	Mapped Meaning { dst, src0, src1, src2 }
21	00000,00000,10100,00000	.0 .20 .0 .0
22	00000,00000,11100,00000	.0 .28 .0 .0
23	11000,00000,11100,00000	.0 .28 .0 .24
24	00100,00000,01000,00000	.0 .8 .0 .4
25	00100,00000,01100,00000	.0 .12 .0 .4
26	01000,00000,01100,00000	.0 .12 .0 .8
27	10000,00000,11000,00000	.0 .24 .0 .16
28	10000,00000,10100,00000	.0 .20 .0 .16
29	01100,00000,00100,00000	.0 .4 .0 .12
30	10100,00000,11100,00000	.0 .28 .0 .20
31	01000,00000,00100,00000	.0 .4 .0 .8

Opcode Encoding

Byte 0 of the 128-bit instruction word contains the opcode. The opcode uses 7 bits. Bit location 7 in byte 0 is reserved for future opcode extension.

The opcodes are encoded and organized into five groups based on the type of operations: Special instructions, move/logic instructions (opcode=00xxxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 110xxxb are reserved.

Note: Opcodes appear in the overall Instruction Set Summary Table as well. The following subsections still serve the purpose of describing various instruction groups.

Move and Logic Instructions

This instruction group has an opcode format of 11xxxxxb.

- The opcodes for move instructions (***mov***, ***sel*** and ***movi***) share the common 5 MSBs in the form of 11000xxb.
- The opcodes for logic instructions (***not***, ***and***, ***or***, and ***xor***) share the common 5 MSBs in the form of 11001xxb.
- The opcodes for shift instructions (***shr***, ***shl***, ***asr***, **ror**, and **rol**) share the common 4 MSBs in the form of 1101xxxb.
- The opcodes for compare instructions (***cmp*** and ***cmpn***) share the common 6 MSBs in the form of 111000xb. Bit 0 indicates whether it is a normal compare, *cmp*, or a special compare-NaN, *cmpn*.

Move and Logic Instructions

Opcode	Instruction	Description	#src	#dst
0x60	nop	No Operation	0	0
0x61	mov	Component-wise move	1	1
0x62	sel	Component-wise selective move based on predication	2	1
0x63	movi	Fast component-wise indexed move	1	1
0x64	not	Component-wise one's complement (bitwise not)	1	1
0x65	and	Component-wise logical AND (bitwise and)	2	1
0x66	or	Component-wise logical OR (bitwise or)	2	1
0x67	xor	Component-wise logical XOR (bitwise xor)	2	1
0x68	shr	Component-wise logical shift right	2	1
0x69	shl	Component-wise logical shift left	2	1
0x6A	smov	Scattered Move	1	1
0x6B	***bfn***	Component-wise boolean function (bitwise)	3	1
0x6C	asr	Component-wise arithmetic shift right	2	1
0x6D	*Reserved*			
0x6E	ror	Component-wise logical rotate right	2	1
0x6F	rol	Component-wise logical rotate left	2	1
0x70	cmp	Component-wise compare, store condition code in destination	2	1
0x71	cmpn	Component-wise compare-NaN, store condition code in destination	2	1
0x72	csel	Component-wise selective move based on result of compare	3	1
0x73-0x76	*Reserved*			
0x77	bfrev	Reverse bits	1	1
0x78	bfe	Bitfield extract	3	1
0x79	bfi1	Bitfield insert macro instruction 1, generate mask	2	1
0x7A	bfi2	Bitfield insert macro instruction 2, insert based on mask	3	1
0x7B-0x7F	*Reserved*			

Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

Flow Control Instructions

Opcode dec	Opcode hex	Instruction	Description	#src	#dst
32	0x20	jmpi	Jump indexed	1	0
33	0x21	brd	Branch - Diverging	1	0
34	0x22	if	If	0/2	0

Opcode dec	Opcode hex	Instruction	Description	#src	#dst
35	0x23	brc	Branch - Converging	1	-
36	0x24	else	Else	1	0
37	0x25	endif	End if	0	0
38	0x26	*Reserved*			
39	0x27	while	While	1	0
40	0x28	break	Break	1	0
41	0x29	cont	Continue	1	0
42	0x2A	halt	Halt	1	0
43	0x2B	calla	Subroutine call absolute	1	1
44	0x2C	call	Subroutine call	1	1
45	0x2D	return	Subroutine return	1	1
46	0x2E	goto	Goto	2	0
47	0x2F	join	Join	2	0

Miscellaneous Instructions

This instruction group has an opcode format of 011xxxxb.

Miscellaneous Instructions

Opcode dec	Opcode hex	Instruction	Description	#src	#dst
49	0x31	send	Send	2	1
53-55	0x35-0x37	*Reserved*			
56	0x38	math	Math functions for extended math pipeline	1/2	1/2
57-63	0x39-0x3F	*Reserved*			

Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

Parallel Arithmetic Instructions

Opcode dec	Opcode hex	Instruction	Description	#src	#dst
64	0x40	add	Component-wise addition	2	1
65	0x41	mul	Component-wise multiply	2	1
66	0x42	avg	Component-wise average of the two source operands	2	1
67	0x43	frc	Component-wise floating point truncate-to-minus-infinity fraction	1	1

Opcode		Instruction	Description	#src	#dst
68	0x44	rndu	Component-wise floating point rounding up (ceiling)	1	1
69	0x45	rndd	Component-wise floating point rounding down (floor)	1	1
70	0x46	rnde	Component-wise floating point rounding toward nearest even	1	1
71	0x47	rndz	Component-wise floating point rounding toward zero	1	1
72	0x48	mac	Component-wise multiply accumulate	2	1
73	0x49	mach	multiply accumulate high	2	1
74	0x4A	lzd	leading zero detection	1	1
75	0x4B	fbh	Find first 1 for UD from msb side, or first 1/0 for D.	1	1
76	0x4C	fbl	First first 1 for UD from lsb side	1	1
77	0x4D	cbit	Count bits set	1	1
78	0x4E	addc	Integer add with carry	2	1 + acc.
79	0x4F	subb	integer subtract with borrow	2	1 + acc.

Vector Arithmetic Instructions

This instruction group has an opcode format of 101xxxxb.

Vector Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
82	0x52	***add3***	Addition ternary	3	1
83	0x53	*Reserved*			
88	0x58	dp4a	Dot product accumulate	3	1
89	0x59	dpas	Dot product accumulate systolic	3	1
90	0x5A	dpasw	Dot product accumulate systolic wide	3	1
91	0x5B	mad (fma)	Component-wise floating point mad computation (a multiple-add)	3	1
92	0x5C	*Reserved*			
93	0x5D	***madm (fmam)***	Component-wise floating point fused multiply and add for macro operations.	3	1
94	0x5E	*Reserved*			
95	0x5F	*Reserved*			

Special Instructions

There are two special instructions, *illegal* and *sync*.

- *Illegal* instruction may be used for instruction padding in memory outside the normal instruction sequence such as before or after the kernel program as well as between subroutines.

The *Synchronize* instruction is used to interact with the software scoreboard synchronize with hardware barriers or the host.

Opcode	Instruction	Description	#src	#dst
0x00	illegal	Illegal instruction	0	0
0x01	sync	Synchronize	1	0
0x02-0x1F	*Reserved*			

Instruction Set Summary Tables

The columns in the following tables specify instruction mnemonics, hex opcodes, full names, instruction groups, processor generation, the number of source operands, whether the instruction supports predication, any support for source modifiers, an indication of supported data types, whether the instruction supports saturation, and any support for conditional modifiers.

See the separate Accumulator Restrictions table for information about how instructions are allowed to use accumulators.

N and Y indicate No (no support for a feature) and Yes (full support for a feature) respectively.

A SrcMod (source modifier) value of Y indicates that a numeric source modifier is allowed, optionally specifying absolute value, negation, or a forced negative value. The value N indicates no source modifier support.

A SrcMod value of ** indicates a logical source modifier is allowed, optionally inverting all source bits (a NOT operation).

In the Src Types and Dst Type columns, Int means any integer type and * means such an extensive list of types that you must refer to the detailed instruction description.

Byte (B, UB) and QWord (Q, UQ) integer types cannot be mixed in the same instruction.

Instruction Set Summary Table A to B (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
add	40	Addition	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
addc	4E	Integer Addition with Carry	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
and	65	Logic And	Move and Logic	2	Y	**	Int	Int	N	Equality only
asr	6C	Arithmetic Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
avg	42	Average	Parallel Arithmetic	2	Y	Y	B, UB W, UW	B, UB W,	Y	Y

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
							D, UD	UW D, UD		
bfe	78	Bit Field Extract	Move and Logic	3	Y	N	UD, D	UD, D	N	N
bfi1	79	Bit Field Insert 1	Move and Logic	2	Y	N	UD, D	UD, D	N	N
bfi2	7A	Bit Field Insert 2	Move and Logic	3	Y	N	UD, D	UD, D	N	N
bfrev	77	Bit Field Reverse	Move and Logic	1	Y	N	UD	UD	N	N
brc	23	Branch Converging	Flow Control	0 or 1	Y	N	D		N	N
brd	21	Branch Diverging	Flow Control	0 or 1	Y	N	D		N	N
break	28	Break	Flow Control	0	Y	N			N	N

Instruction Set Summary Table C to E (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
call	2C	Call	Flow Control	0	Y	N		D, UD	N	N
calla	2B	Call Absolute	Flow Control	0	Y	N		D, UD	N	N
cbit	4D	Count Bits Set	Move and Logic	1	Y	N	UB, UW, UD	UD	N	N
cmp	70	Compare	Move and Logic	2	Y	Y	*	*	N	Y
cmpn	71	Compare NaN	Move and Logic	2	Y	Y	*	*	N	Y
cont	29	Continue	Flow Control	0	Y	N			N	N
csel	72	Conditional Select	Move and Logic	3	N	Y	F	F	Y	N
dp4a	58	Dot Product 4 Accumulate	Vector Arithmetic	2	Y	Y	F	F	Y	Y
else	24	Else	Flow Control	0	N	N			N	N
endif	25	End If	Flow Control	0	N	N			N	N

Instruction Set Summary Table F to L (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
fbh	4B	Find First Bit from MSB Side	Move and Logic	1	Y	N	D, UD	UD	N	N
fbl	4C	Find First Bit from LSB Side	Move and Logic	1	Y	N	UD	UD	N	N
frc	43	Fraction	Parallel Arithmetic	1	Y	Y	F	F	N	Y
goto	2E	Goto	Flow Control	0	Y	N			N	N
halt	2A	Halt	Flow Control	0	Y	N			N	N
if	22	If	Flow Control	0	Y	N			N	N
illegal	00	Illegal	Special	0	N	N			N	N
jmpi	20	Jump Indexed	Flow Control	1	Y	N	D		N	N
join	2F	Join	Flow Control	0	Y	N			N	N
line	59	Line	Vector Arithmetic	2	Y	Y	F	F	Y	Y
lzd	4A	Leading Zero Detection	Move and Logic	1	Y	Y	D, UD	UD	Y	Y

Instruction Set Summary Table M to P (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
mac	48	Multiply Accumulate	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
mach	49	Multiply Accumulate High	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
mad	5B	Multiply Add	Vector Arithmetic	3	Y	Y	*	*	Y	N
madm	5D	Multiply Add for Macro	Vector Arithmetic	3	Y	Y	*	*	Y	N
math	38	Extended Math Function	Miscellaneous	2	Y	N	*	*	Y	N
mov	01	Move	Move and Logic	1	Y	Y	*	*	Y	Y
mov	61	Move	Move and Logic	1	Y	Y	*	*	Y	Y
movi	63	Move Indexed	Move and Logic	1	Y	Y	*	*	Y	N
mul	41	Multiply	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
nop	60	No Operation	Move and Logic	0	N	N			N	N

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
not	64	Logic Not	Move and Logic	1	Y	**	Int	Int	N	Equality only
or	66	Logic Or	Move and Logic	2	Y	**	Int	Int	N	Equality only

Instruction Set Summary Table R to X (Listed by Instruction Mnemonic)

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
ret	2D	Return	Flow Control	1	Y	N	D, UD		N	N
rndd	45	Round Down	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rnde	46	Round to Nearest or Even	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rndu	44	Round Up	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
rndz	47	Round to Zero	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
sad2	50	Sum of Absolute Difference 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
sada2	51	Sum of Absolute Difference Accumulate 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
sel	62	Select	Move and Logic	2	Y	Y	*	*	Y	Y
send	31	Send Message	Miscellaneous	2	Y	N	*	*	N	N
sendc	32	Send Message Conditional	Miscellaneous	2	Y	N	*	*	N	N
shl	69	Shift Left	Move and Logic	2	Y	Y	Int	Int	Y	Y
shr	68	Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
smov	6A	Scattered Move	Move and Logic	1	Y	N	*	*	N	N
subb	4F	Integer Subtraction with Borrow	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
sync	01	Synchronize	Special	1	Y	N	*	*	N	N
while	27	While	Flow Control	0	Y	N			N	N

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
xor	67	Logic Xor	Move and Logic	2	Y	**	Int	Int	N	Equality only

Accumulator Restrictions

This section describes restrictions on accumulator access: general restrictions, restrictions for specific instructions, and how those specific restrictions vary for processor generations. See Accumulator Registers

for a description of the accumulator registers.

Accumulator registers can be accessed as explicit source or destination operands, as an implicit source value when specified for a particular instruction (*sada2* for example), and as an implicit destination when the AccWrEn instruction option is used.

These general rules apply to accumulator access:

1. Flow control, *send*, *sendc*, and *wait* instructions cannot use accumulators.
2. Instructions that specify an implicit accumulator destination (with AccWrEn) cannot specify an explicit accumulator destination operand.
3. Accumulator may be used in one of Src0 or Src1 only for MUL/MAC instruction.
4. Extra accumulator precision for integer is only supported on Src0.
5. Src2 cannot use accumulator for 3 source instructions.
6. INT2FLT format conversion cannot use accumulator as source.

These descriptions are frequently used in this table:

- No restrictions.
- No accumulator access, implicit or explicit.
- Source operands cannot be accumulators.
- Source modifier is not allowed if source is an accumulator.
- Accumulator is an implicit source and thus cannot be an explicit source operand.
- Accumulator cannot be destination, implicit or explicit.
- AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.

These minor cases occur occasionally in the table:

- Integer source operands cannot be accumulators.
- No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.

- An accumulator can be a source or destination operand but not both.

A few instructions use more than one of the listed restrictions.

Indirect address access is not allowed when dst is accumulator.

Example - mul (16|M0) acc0.0<1>:w -r[a0.12,-256]<8,0>:w r39.10<32;16,0>:w

Accumulator Restrictions

Instructions	Accumulator Restrictions
add	No restrictions.
add3	No restrictions.
addc	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand. Accumulator is not allowed to be explicit source.
and	Source modifier is not allowed if source is an accumulator.
asr *avg*	No restrictions.
bfe *bfi1* *bfi2* *bfrev* *cbit*	No accumulator access, implicit or explicit.
bfn	No restrictions.
cmp	No restrictions.
cmpn	No restrictions.
csel	No restrictions.
fbh *fbl*	No accumulator access, implicit or explicit.
frc	No restrictions.
line	Source operands cannot be accumulators.
lzd	Accumulator cannot be source, implicit or explicit.
mac	Integer source operands cannot be explicit accumulators.
mach	Accumulator is an implicit source and thus cannot be an explicit source operand. AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand.

Instructions	Accumulator Restrictions
mad	Integer source operands cannot be accumulators.
madm	No restrictions.
math	No accumulator access, implicit or explicit.
movi	Source operands cannot be accumulators.
mul	Integer source operands cannot be accumulators.
not *or*	Source modifier is not allowed if source is an accumulator.
rndd *rnde* *rndu* *rndz*	No restrictions.
rol	No restrictions.
ror	No restrictions.
sel	No restrictions.
shl	Accumulator cannot be destination, implicit or explicit.
shr	No restrictions.
smov	No restrictions.
subb	AccWrEn is required. The accumulator is an implicit destination and thus cannot be an explicit destination operand. Accumulator is not allowed to be explicit source.
xor	Source modifier is not allowed if source is an accumulator.

Native Instruction BNF

Describes Syntax supported by IGA.

Grammar

IGA Grammar

The following describes a super-set of the legal inputs to IGA. Many constructs below contain additional restrictions not represented syntactically (e.g. types on instructions).

IGA Grammar			
A program is a sequence of blocks, where the first block may not have a label			
Program	->	(*LabelDefLine*? *Instruction** (*LabelDefLine Instruction*))?	
LabelDefLine	->	*LabelDef LineTerminator*	
LabelRef	->	*Ident*	
LabelDef	->	*Ident* `:`	
InstructionLine	->	*Instruction LineTerminator*	
LineTerminator	->	`\n`	
	|	*EOF*	end of file
The rough layout of an instruction			
Instruction	->	*PredWrEn*? *Mnemonic ExecInfo FlagModifier*? *Operands InstOptions*?	
	|	*LoadStoreInstruction*	Confer with the page on load/store syntax for these.
where			
Mnemonic	->	*Ident*	E.g. `mov` or `add`
	|	`math` . *MathFC*	E.g. math.inv
	|	`send` . *SFID*	E.g. send.dc0
	|	`bfn` . *BfnFC*	E.g. bfn.0xE7
	|	`dpas` . *DpasFC*	
e.g. sqt in math.sqt ... math function controls extend the opcode; e.g. math.inv ...			
MathFC	->	`inv`	
	|	`log`	
	|	`exp`	
	|	`sqt`	

IGA Grammar			
	|	rsqt	
	|	pow	
	|	sin	
	|	cos	
	|	fdiv	
	|	invm	
	|	rsqtm	
e.g. dco in send.dco ... send function id determines what send type it is			
SFID	->	dc0	
	|	dc1	
	|	dc2	
	|	dcr0	
	|	gtwy	
	|	null	
	|	pixi	
	|	rc	
	|	smpl	
	|	ts	
	|	urb	
	|	vme	
	|	cre	
	|	ugm	
	|	tgm	
	|	slm	
	|	btd	
	|	rta	
BfnFC	->	*See BooleanFuncCtrl enumeration for a list of legal symbols*	
DpasFC	->	*See the SystolicFC enumeration for a list of legal subop symbols*	
Execution mask part e.g. (16|M16)			
ExecInfo	->	(*ExecWidth* | *EMaskOff*)	
where			
ExecWidth	->	1 | 2 | 4 | 8 | 16 | 32	
the emask			

IGA Grammar			
EMaskOff	->	`M0` | `M4` | … | `M28`	
Instruction write-enable (formly NoMask) as well as predication Examples: (W), (f1.0), or (W~f1.1)			
PredWrEn	->	`(` *WrEn* `)` | `(` *Pred* `)` | `(` *WrEn* `&` *Pred* `)`	
where			
WrEn	->	`W`	
Pred	->	*PredNegate*? *FlagRegRef PredMask*?	
PredNegate	->	`~`	
see PredCtrl enum 3D-Media-GPGPU Engine EU Overview ISA Introduction Instruction Set Reference EUISA Enumerations **PredCtrl**			
PredMask	->	`.` *PredFunc*	
where			
PredFunc	->	`anyv`	
	|	`allv`	
	|	`any2h`	
	|	`all2h`	
	|	`any4h`	
	|	`all4h`	
	|	`any8h`	
	|	`all8h`	
	|	`any16h`	
	|	`all16h`	
	|	`any32h`	
	|	`all32h`	
A condition modifier indicates the flag register should be updated with respect to the given relational function. E.g. (le)f0.0			
FlagModifier	->	`(` *Func* `)` *FlagRegRef*	
where			
Func	->	`lt`	less than
	|	`le`	less than or equal
	|	`gt`	greater than

IGA Grammar			
	|	ge	greater than or equal
	|	eq	equal
	|	ne	not equal
	|	ov	overflow
	|	un	unordered (NaN)
	|	lz	less than 0; only supported on sel instructions
	|	gz	greater than 0; only supported on sel instructions
Operands	->	*OperandsEmpty*	e.g. nop, illegal, ...
	|	*OperandsUnary*	e.g. mov, not, ...
	|	*OperandsBinary*	e.g. and, add, mul, ..
	|	*OperandsTernary*	e.g. mad, csel, bfn, dpas, ...
	|	*OperandsSingleSource*	e.g. wait, jmpi, ...
	|	*OperandsDoubleSource*	e.g. break, cont, if, else (branch control is a sub function)
	|	*OperandsSend*	sends or sendsc
OperandsEmpty	->		
OperandsUnary	->	*DstOperand SrcOperand*	
OperandsBinary	->	*DstOperand SrcOperand SrcOperand*	
OperandsTernary	->	*DstOperand SrcOperand SrcOperand SrcOperand*	
OperandsSingleSource	->	*SrcOperand*	
OperandsDoubleSource	->	*SrcOperand SrcOperand*	

IGA Grammar			
OperandsSend	->	*DstOperand SrcOperand SrcOperand ExDescOp DescOp*	
ExDescOp	->	*AddrRegRef* | *IntLit*	
DescOp	->	*AddrRegRef* | *IntLit*	
Destination operands			
DstOperand	->	*DstOpDirect*	direct register access; e.g. r14.3<1> or sr0.1
	|	*DstOpIndirect*	indirect access; e.g. r[a0.4,16]<1>
where			
DstOpDirect	->	*DstModif*? *Register DstRegion*? *DstType*?	
DstOpIndirect	->	*DstModif*? `r [` *AddrRegName* `.` `]` *DstRegion*?	
DstModif	->	`(sat)`	saturation modifier
DstRegion	->	`<` *DstHzStride* `>`	
where			
DstHzStride	->	`1` | `2` | `4`	
Source operands -- Semantic checks for regioning disallow regions on certain registers (e.g. cannot region ce)			
SrcOperand	->	*SrcOpDirect*	direct register access; e.g. r13.0<8;8,1>
	|	*SrcOpIndirect*	indirect register access; e.g. r[a0.0,16]<1,0>
	|	*SrcOpImm*	source immediate operand; e.g. 13:ud, 0x155:q, 1.2:f
where			
SrcOpDirect	->	*SrcModifier*? *Register SrcRegion*? *SrcRegType*?	
SrcOpIndirect	->	*SrcModifier*? `r [` *IndexReg* `(` `,` *ConstExpr*`)` `]` *SrcRegion*? *SrcRegType*?	

IGA Grammar			
SrcOpImm	->	*ConstExpr SrcImmType*	
We semantically enforced that ~ applies only to bitwise ops (e.g. and, or, ...)			
SrcModifier	->	~	bitwise negation for bitwise ops (one's complement)
	|	-	signed negation (two's complement)
	|	(abs)	absolute value
	|	- (abs)	negation of the absolute value
the source region			
SrcRegion	->	*SrcRegionVWH* | *SrcRegionWH* | *SrcRegionVH* | *SrcRegionH*	
SrcRegionVWH	->	< *VtStride* ; *Width* , *HzStride* >	
SrcRegionWH	->	< *Width* , *HzStride* >	for indirect only (uses VxH/Vx1 mode)
SrcRegionVH	->	< *VtStride* ; *HzStride* >	ternary src0 or src1 region
SrcRegionH	->	< *HzStride* >	ternary src2 region
VtStride	->	1	
	|	2	
	|	4	
	|	8	
	|	16	
	|	32	the number of elements to advance the row after stepping Width times
Width	->	1	
	|	2	
	|	4	
	|	8	
	|	16	the number elements per row (before advancing vert stride)
HzStride	->	0	
	|	1	

IGA Grammar			
	|	2	
	|	4	the number of elements to step horizontally between channels
Subregisters are in units of the type of the operand they are applied to. That is 8-byte types only permit subregisters 0-3, 4-byte types permit 0-7, and so forth.			
Register	->	*RegName* (. *SubReg*)?	
where			
See the EU section on ARF registers			
RegName	->	*Names*	
where			
Names	->	*AddrRegName*	address register
	|	acc0	
	|	…	
	|	acc11	up to acc1 and acc9
	|	*FlagRegName*	e.g. f0 or f1
	|	fc0	
	|	…	
	|	fc4	flow control registers
	|	ce	channel enable register
	|	cr0	
	|	ip	instruction pointer
	|	n0	
	|	n1	
	|	msg0	
	|	…	
	|	msg31	
	|	null	the null register
	|	r0	general registers
	|	…	
	|	r255	
	|	sr0	
	|	sr1	
	|	tdr	

IGA Grammar			
	|	tm0	
The subregister. Various registers support different number of subregisters. Subregister units are in terms of data type size. E.g. up to: .31 for :b, .15 for :w, .7 for :d, and .3 for :q etc ...			
SubReg	->	*IntLit*	
A reference to a flag register. Examples: f0.0 or f1.1			
FlagRegRef	->	*FlagRegName* (. *FlagSubReg*)?	
where			
FlagSubReg	->	0 | 1	
FlagRegName	->	f0 … f3	
An address register reference; e.g. a0.3 (sometimes called the "index register")			
AddrRegRef	->	*AddrRegName* (. *AddrSubReg*)?	
where			
AddrSubReg	->	0 | … | 15	
AddrRegName	->	a0	
Destination types are the allowable types on registers			
DstType	->	*RegType*	
Source types are the same as the destination types with the inclusion of the packed vector types.			
SrcImmType	->	*ImmType*	
SrcRegType	->	*RegType*	
Register and immediate operand types. RegTypes includes non-vector immediates.			
RegType	->	:b	8b signed integer
	|	:ub	8b unsigned integer

IGA Grammar			
	|	:w	16b signed integer
	|	:uw	16b unsigned integer
	|	:d	32b signed integer
	|	:ud	32b unsigned integer
	|	:q	64b signed integer
	|	:uq	64b unsigned integer
	|	:hf	IEEE-754 16b float
	|	:bf	16b bfloat
	|	:f	IEEE-754 32b float
	|	:df	IEEE-754 64b float
Immediate Operand Types			
ImmType	|	:uv	8x4b packed unsigned int
	|	:v	8x4b packed signed int
	|	:w	16b signed integer
	|	:uw	16b unsigned integer
	|	:d	32b signed integer
	|	:ud	32b unsigned integer
	|	:q	64b signed integer
	|	:uq	64b unsigned integer
	|	:vf	4x8b packed float
	|	:hf	IEEE 754 16b float
	|	:bf	16b bfloat
	|	:f	IEEE 754 32b float
	|	:df	IEEE 754 64b float
Expressions. Constant expressions are evaluated by the assembler with host semantics.			
ConstExpr	->	*IntLit*	integer literal
	|	*FltLit*	floating point literal
	|	*Ident*	reference to a symbolic constant
	|	*UnOp ConstExpr*	unary constant expressions (e.g. ~((1NUM_BITS)-1)
	|	*ConstExpr BinOp ConstExpr*	binary constant expressions e.g. 1 + 2
	|	(*ConstExpr*)	constant grouping expression (1 + 2)
	|	*LabelRef*	a reference to a label

IGA Grammar			
			definition
Unary operators			
UnOp	->	~	one's complement
	|	-	two's complement
Binary operators (in precedence order)			
BinOp	->	*	multiplicative operators
	|	/	
	|	%	
	|	+	additive operators
	|	-	
	|	<<	bit-shift operators
	|	>>	
	|	>>>	shift with zero-fill
	|	&	bitwise (logical) operators
	|	^	
	|	|	
InstOptions	->	{ (*InstOpt* (, *InstOpt*)*)? }	
where			
InstOpt	->	AccWrEn	
	|	Atomic	
	|	Breakpoint	
	|	Compacted	
	|	EOT	
	|	*DepInfo*	
DepInfo	->	RegDist | SBIDAlloc | SBIDWaitSrc | SBIDWaitDst	
where			

IGA Grammar			
RegDist	->	`@`*IntLit*	minimum distance on the current pipe
	|	`F@`*IntLit*	minimum distance on the float pipe
	|	`I@`*IntLit*	minimum distance on the integer pipe
	|	`L@`*IntLit*	minimum distance on the long pipe
SBIDAlloc	->	`$`*IntLit*	allocate a dependency barrier (SBID: (S)core(B)oard (ID)
SBIDWaitSrc	->	`$`*IntLit*`.src`	wait until sources are read (on the instruction that allocated the SBID)
SBIDWaitDst	->	`$`*IntLit*`.dst`	wait until writeback completes (of the instruction that allocated the SBID)
Identifiers			
Ident	->	`[A-Za-z_][A-Za-z0-9_]*`	
An integer literal. For each, _ is an optional lexical separator for readability			
IntLit	->	*IntLitDec*	decimal literal e.g. 13, 14, 4000, or 4_000
	|	*IntLitHex*	hex literal e.g. 0xFE or 0xABCD_EFGH
	|	*IntLitBin*	binary literal e.g. 0b1011 would be 11
where			
IntLitDec	->	`[0-9][_0-9]*`	
IntLitHex	->	`0x[0-9a-fA-F]?[_0-9a-fA-F]*`	
IntLitBin	->	`0b[01][_01]*`	
A floating point literal			
FltLit	->	*FltLitDec*	decimal 3.1414
	|	*FltLitSci*	scientific 6.02e23 or 6.67384e-11

IGA Grammar			
	|	`inf`	special symbol for infinity (0x...)
	|	`nan`	special symbol for not a number
where			
FltLitDec	->	`([0-9][_0-9]*)?.[0-9][_0-9]*|([0-9][_0-9]*)?.([0-9][_0-9]*)?`	
FltLitSci	->	`([0-9][_0-9]*)?.[0-9][_0-9]*e([0-9][_0-9]*)`	

Load-Store Pseudo Instructions

The assembler may map certain send instructions to higher-level load/store pseudo-instructions for readability in assembly. For each there's an underlying send instruction. For example, the following two instructions generate identical ISA.

```
load.ugm.d32.a64 (32|M0) r10:2 [r20:4]
send.ugm (32|M0) r10 r20 null:0 0x0 0x08200580
```

The grammar for supported pseudo instructions is given below.

NOTE: The syntax below does not capture or enforce all hardware restrictions; these may vary per platform. Consult the **Data Port** pages for restrictions. Secondly, not all instructions listed in that section are mapped to this sytnax.

Terminal/Nonterminal	Expands to
Top-level load instructions (loads, stores, atomics, and control IO instructions)	
LdStInst ?	LdInst | StInst | AtInst
LdInst ?	Predication? LdOpts ExecInfo DataOperand AddrOperand
StInst ?	Predication? StOpts ExecInfo AddrOperand DataOperand
AtInst ?	Predication? AtOpts ExecInfo DataOperand AddrOperand DataOperand
Load/Store Controls	
LdOptions ?	.DataSize .AddrSize LdCaching
StOptions ?	.DataSize .AddrSize StCaching
AtOptions ?	.DataSize .AddrSize AtCaching
DataSize ?	d (8 | 16 | 32 | 64) (VecSize? t? | XYZWMask) | d8u32 | d16u32 • data sizes map to **DP_DATA_SIZE** • A suffix of t means **DP_TRANSPOSE**
VecSize ?	x1 | x2 | x3 | x4 | x8 | x16 | x32 | x64 • maps to **DP_VECT_SIZE**

Terminal/Nonterminal	Expands to
XYZWMask ?	. x? y? z? w? • maps to **DP_CMASK**
AddrSize ?	a16 | a32 | a64 • maps to **DP_ADDR_SIZE**
LdCaching ?	An optional entry from **DP_CACHE_LOAD** • If omitted this defaults to .df.df
StCaching ?	An optional entry from **DP_CACHE_STORE** • If omitted this defaults to .df.df
AtCaching ?	An optional entry from **DP_CACHE_STORE** (a subset of store options work for atomics) • If omitted this defaults to .df.df
Operands	
DataOperand ?	(GrfReg | null) : Int? a register followed by an optional payload length
AddrOperand ?	AddrSurfType ? [GrfReg (+ Int)?] • If AddrSurfType then defaults it defaults to flat
AddrSurfType ?	flat | bti Surf | bss Surf | ss Surf • examples: flat, bti[0x1], bss[a0.4] • this field maps to **DP_ADDR_SURFACE_TYPE**
Surf ?	[Int | a0. Int]

Examples

load.ugm.d32.a64 (32|M0) r10:2 [r20:4] a SIMT32 load of 32b (d32) data elements into r10..r11 (r10:2) using 64b addresses from r20..23 and the FLAT address type (c.f. **DP_ADDR_SURFACE_TYPE**); caching uses the default state settings (.df.df)
load.ugm.d32.a64.uc.uc (32|M0) r10:2 [r20:4] the same as above except using uncached (**DP_CACHE_LOAD**'s L1UC_L3UC value)
store.slm.d32x4.a32 (32|M0) [r20:2] r10:8 a SIMT32x4 SLM store (128 elements / OpenCL float4) of data from r10..r17 (r10:8) using 32b addresses from r20..23
atomic_fadd.ugm.d32.a64 (16|M0) null [r20:2] r10:1 a SIMT16 atomic floating-point add with no return payload to stateless untyped memory (.ugm and flat address type)
load.ugm.d32x16t.a32 (1|M0) r10:1 bti[0x2][r20:1] a SIMD1 load of 16 x 32b data elements into r10 from the surface pointed to by BTI index 2

load.ugm.d32x16t.a32 (1|M0) r10:1 bss[a0.4][r20:1] a SIMD1 load of 16 x 32b data elements into r10 from the surface object pointed to by a0.4

Load Mnemonics

Op	Maps to Descriptor
load	DP_LOAD
load_quad	DP_LOAD_CMASK

Store Mnemonics

Op	Maps to Descriptor
store	DP_STORE
store_quad	DP_STORE_CMASK

Atomic Mnemonics

Op	Maps to Descriptor
atomic_load	DP_ATOMIC_LOAD
atomic_store	DP_ATOMIC_STORE
atomic_and	DP_ATOMIC_AND
atomic_xor	DP_ATOMIC_XOR
atomic_or	DP_ATOMIC_OR
atomic_iinc	DP_ATOMIC_INC
atomic_idec	DP_ATOMIC_DEC
atomic_iadd	DP_ATOMIC_ADD
atomic_isub	DP_ATOMIC_SUB
atomic_icas	DP_ATOMIC_CMPXCHG
atomic_smin	DP_ATOMIC_MIN
atomic_smax	DP_ATOMIC_MAX
atomic_umin	DP_ATOMIC_UMIN
atomic_umax	DP_ATOMIC_UMAX
atomic_fadd	DP_ATOMIC_FADD
atomic_fsub	DP_ATOMIC_FSUB
atomic_fmin	DP_ATOMIC_FMIN
atomic_fmax	DP_ATOMIC_FMAX
atomic_fcas	DP_ATOMIC_FCMPXCHG

Instruction Set Reference

This chapter describes the functions of 3D Media GPGPU Execution Units, listed in alphabetical order according to assembly language mnemonic.

EUISA Instructions List

Conventions

This section describes conventions used in instruction reference pages.

For each instruction that has source or destination types, a table lists the allowed type combinations and may also indicate the processor generations that support certain combinations. A notation like *W indicates that UW and W are both allowed. Multiple types listed together mean that any combination (Cartesian product) of the listed types is allowed.

If a source operand is floating-point, all source operands must have the same floating-point data type.

The Q and UQ types cannot be mixed with the B or UB types, neither as different source types nor as source type and destination type.

Accumulator restrictions are described in the Accumulator Restrictions section and also appear in instruction descriptions.

Pseudo Code Format

Instructions are explained in the following pseudo-code format that resembles the assembly instruction format.

```
[(pred)] opcode (exec_size) dst src0 [src1]
```

Square brackets "[ ]" indicate that a field is optional. Saturation modifiers and instruction options are omitted for simplicity.

General Macros and Definitions

INST_MIN_SIZE is defined as a constant of 8 bytes.

```
#define INST_MIN_SIZE  8  // Instruction minimum size in bytes (for the compact instruction format)
```

The floor function converts a floating-point value to an integral floating point value. For a given floating point value, from its closest two integral float values, floor returns the one that is closer to negative infinity. For example, floor(1.3f) = 1.0f and floor(-1.3f) = -2.0f.

```
float floor(float g)
{
    return maximum(any integral float f: f <= g)
}
```

The Condition function takes the conditional signals {SN, ZR, OF, IN, NC} of result, generates a Boolean value according to a conditional evaluation controlled by the conditional modifier cmod, and returns the Boolean.

```
Bool Condition(result, cmod)
```

The ConditionNaN function takes the conditional signals {SN, ZR, OF, IN, NC, NS} of result, generates a Boolean value according to a conditional evaluation controlled by the conditional modifier cmod, and returns the Boolean. The only difference between Condition and ConditionNaN is that ConditionNaN uses the NS (NaN of the second source) signal.

```
Bool ConditionNaN(result, cmod)
```

The Jump function jumps the instruction sequence from the current instruction location by InstCount 8-byte units, where each 16-byte native instruction is two units and each 8-byte compact instruction is one unit. If InstCount is positive and greater than zero, is an unconditional jump forward. If InstCount is negative, is an unconditional jump backward. If InstCount is zero, IP stays on the current instruction in an infinite loop.

```
void Jump(int InstCount)
{
    IP = IP + (InstCount * INST_MIN_SIZE)
}
```

Evaluate Write Enable

The WrEn should be evaluated as below.

Note: MaskCtrl = NoMask (1) skips the check for PcIP[n] == ExIP before enabling a channel.

```
if ( MaskCtrl == 1 ) {
  for ( n = 0; n < exec_size; n++ ) {
    WrEn[n] = 1;
  }
}
else {
  for ( n = 0; n < exec_size; n++ ) {
    if ( PcIP[n] == ExIP ) {
      WrEn[n] = 1;
    }
    else {
      WrEn[n] = 0;
    }
  }
}

if ( PredCtrl != 0000b ) {
  for ( n = 0; n < exec_size; n++ ) {
    WrEn[n] = WrEn[n] & PMask[n];
  }
}

for ( n = exec_size; n < 32; n++ ) {
  WrEn[n] = 0;
}
```

EUISA Instructions

Symbol	Name	Source
add	**Addition**	EUISA
add3	**Addition Ternary**	EUISA
addc	**Addition with Carry**	EUISA
asr	**Arithmetic Shift Right**	EUISA
avg	**Average**	EUISA
bfe	**Bit Field Extract**	EUISA
bfi1	**Bit Field Insert 1**	EUISA
bfi2	**Bit Field Insert 2**	EUISA
bfrev	**Bit Field Reverse**	EUISA
bfn	**Boolean Function**	EUISA
brc	**Branch Converging**	EUISA
brd	**Branch Diverging**	EUISA
break	**Break**	EUISA
call	**Call**	EUISA
calla	**Call Absolute**	EUISA
cmp	**Compare**	EUISA
cmpn	**Compare NaN**	EUISA
csel	**Conditional Select**	EUISA
cont	**Continue**	EUISA
cbit	**Count Bits Set**	EUISA
dpas	**Dot Product Accumulate Systolic**	EUISA
dpasw	**Dot Product Accumulate Systolic Wide**	EUISA
else	**Else**	EUISA
endif	**End If**	EUISA
math	**Extended Math Function**	EUISA
fbl	**Find First Bit from LSB Side**	EUISA
fbh	**Find First Bit from MSB Side**	EUISA
frc	**Fraction**	EUISA
goto	**Goto**	EUISA
halt	**Halt**	EUISA
if	**If**	EUISA
illegal	**Illegal**	EUISA
join	**Join**	EUISA
jmpi	**Jump Indexed**	EUISA

Symbol	Name	Source
lzd	**Leading Zero Detection**	EUISA
and	**Logic And**	EUISA
not	**Logic Not**	EUISA
or	**Logic Or**	EUISA
xor	**Logic Xor**	EUISA
mov	**Move**	EUISA
movi	**Move Indexed**	EUISA
mul	**Multiply**	EUISA
mac	**Multiply Accumulate**	EUISA
mach	**Multiply Accumulate High**	EUISA
mad	**Multiply Add**	EUISA
nop	**No Operation**	EUISA
ret	**Return**	EUISA
rol	**Rotate Left**	EUISA
ror	**Rotate Right**	EUISA
rndd rnde rndu rndz	**Round Instructions** ▪ **Round Down** ▪ **Round to Nearest or Even** ▪ **Round Up** ▪ **Round to Zero**	EUISA
sel	**Select**	EUISA
send	**Send Message**	EUISA
sendc	**Send Message Conditional**	EUISA
shl	**Shift Left**	EUISA
shr	**Shift Right**	EUISA
subb	**Integer Subtraction with Borrow**	EUISA
sync	**Synchronize**	EUISA
while	**While**	EUISA

Round Instructions

rndd - Round Down

rndu - Round Up

rnde - Round to Nearest or Even

rndz - Round to Zero

rndd - Round Down

Description:

The *rndd* instruction takes component-wise floating point downward rounding (to the integral float number closer to negative infinity) of src0 and storing the rounded integral float results in dst. This is commonly referred to as the floor() function.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round Down in IEEE mode

src0	-inf	-finite	-denorm	-0	+0	+denorm	+finite	+inf	NaN
dst	-inf	-finite	^	-0	+0	+0	**	+inf	NaN
Notes:									
^	Note: {-1, -0} depending on the Single Precision Denorm Mode.								
**	Result may be {+finite, +0}.								

Floating-Point Round Down in ALT mode

src0	-fmax	-finite	-denorm	-0	+0	+denorm	+finite	+fmax	***
dst	-fmax	-finite	-0	-0	+0	+0	**	+fmax	
Notes:									
**	Result may be {+finite, +0}.								
***	Result is undefined if src0 is {-inf, +inf, NaN}.								

rnde - Round to Nearest or Even

Description:

The *rnde* instruction takes component-wise floating-point round-to-even operation of src0 with results in two pieces - a downward rounded integral float results stored in dst and the round-to-even increments stored in the rounding increment bits. The round-to-even increment must be added to the results in dst to create the final round-to-even values to emulate the round-to-even operation, commonly known as the round() function. The final results are the one of the two integral float values that is nearer to the input values. If neither possibility is nearer, the even alternative is chosen.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round to Nearest or Even in IEEE mode

src0	-inf	-finite	-denorm	-0	+0	+denorm	+finite	+inf	NaN
dst	-inf	*	-0	-0	+0	+0	**	+inf	NaN
Notes:									
*	Result may be {-finite, -0}.								
**	Result may be {+finite, +0}.								

Floating-Point Round to Nearest or Even in ALT mode

src0	-fmax	-finite	-denorm	-0	+0	+denorm	+finite	+fmax	***
dst	-fmax	*	-0	-0	+0	+0	**	+fmax	
Notes:									
*	Result may be {-finite, -0}.								
**	Result may be {+finite, +0}.								
***	Result is undefined if src0 is {-inf, +inf, NaN}.								

rndu - Round Up

Description:

The *rndu* instruction takes component-wise floating point upward rounding (to the integral float number closer to positive infinity) of src0, commonly known as the ceiling() function.

Each result follows the rules in the following tables based on the floating-point mode.

Floating-Point Round Up in IEEE mode

src0	-inf	-finite	-denorm	-0	+0	+denorm	+finite	+inf	NaN
dst	-inf	*	-0	-0	+0	^	+finite	+inf	NaN
Notes:									
*	Result may be {-finite, -0}.								
^	Note: {+1, +0} depending on the Single Precision Denorm Mode.								

Floating-Point Round Up in ALT mode

src0	-fmax	-finite	-denorm	-0	+0	+denorm	+finite	+fmax	***
dst	-fmax	*	-0	-0	+0	+0	+finite	+fmax	
Notes:									
*	Result may be {-finite, -0}.								
***	Result is undefined if src0 is {-inf, +inf, NaN}.								

rndz - Round to Zero

Description:

The *rndz* instruction takes component-wise floating-point round-to-zero operation of src0 with results in two pieces - a downward rounded integral float results stored in dst and the round-to-zero increments stored in the rounding increment bits. The round-to-zero increment must be added to the results in dst to create the final round-to-zero values to emulate the round-to-zero operation, commonly known as the truncate() function. The final results are the one of the two closest integral float values to the input values that is nearer to zero.

Floating-Point Round to Zero in IEEE mode

src0	-inf	-finite	-denorm	-0	+0	+denorm	+finite	+inf	NaN
dst	-inf	*	-0	-0	+0	+0	**	+inf	NaN
Notes:									
*	Result may be {-finite, -0}.								
**	Result may be {+finite, +0}.								

Floating-Point Round to Zero in ALT mode

src0	-fmax	-finite	-denorm	-0	+0	+denorm	+finite	+fmax	***
dst	-fmax	*	-0	-0	+0	+0	**	+fmax	
Notes:									
*	Result may be {-finite, -0}.								
**	Result may be {+finite, +0}.								
***	Result is undefined if src0 is {-inf, +inf, NaN}.								

srnd - Stochastic Rounding

Description:

The srnd instruction takes component-wise source data and performs rounding with a random number. This is special rounding mode and no industry standard defining the implementation details. The key of this rounding algorithm is to add a random number to the source data mantissa and get an intermedia

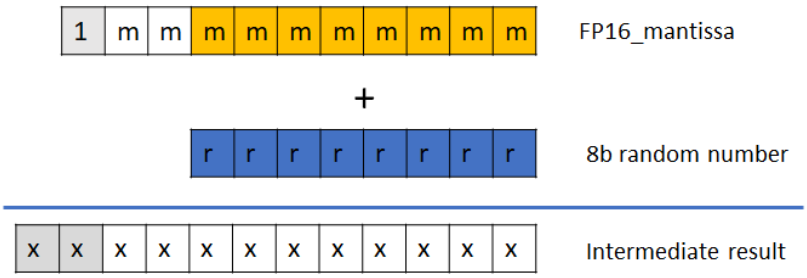
result. The intermediate result mantissa will be normalized and truncate to get detonation mantissa and exponent.

The random number is provided in instruction as src1. **Our ISA doesn't support mixed data type of integer and float-point for sources. So we define the src1 as floating-point data type, but HW takes it as an integer number and truncates the desired width bits from the number. The user can put the random number in Src1 without any data type conversion.** The bits used for different data type are shown as follow.

src0	Des	Random Number in Src1	Random number Used
HF	BF8	16-bits	8-bits (src1[7:0])
F	HF	32-bits	13-bits (src1[12:0])

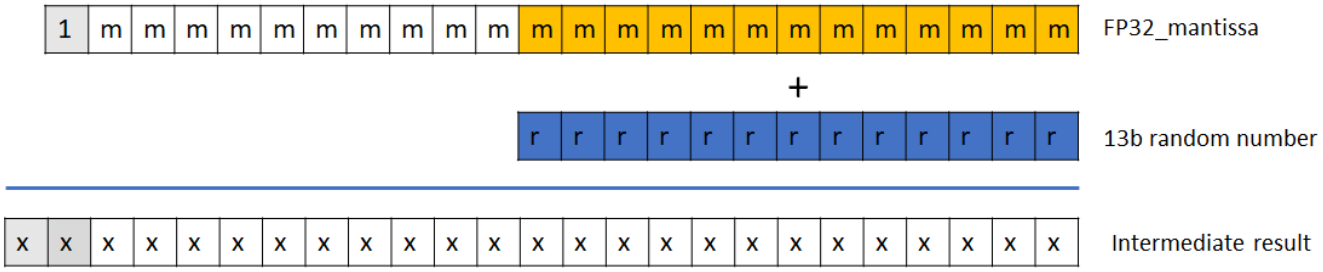
HF->BF8 conversion

The HF's mantissa is added by an 8-bit random number to get intermediate result.



F->HF conversion

The F's mantissa is added by a 13-bit random number to get intermediate result.



Although stochastic rounding is not IEEE standard, our implementation follows the IEEE convention as close as possible. The followings are the pseudo codes.

```
if(Input is sNaN){
    output Quieted_NaN(input);
}
else if(Input is qNaN){
    output destination qNaN
```

```
}
else if(Input is Inf){
    output destination Inf
}
else if(unbiasedExponent(Input) < (-14))
{
     //The input value is in denormal range for FP16 format
     //Output may be the minimal normal after rounding which is compliance with standard RNE.
     Apply RNE to Input
}
else
{
    //Stochastic rounding
    Step 1:  Extract the random number and align to the input mantissa (explained above)
    Step 2:  Perform the addition of input mantissa and random number
    Step 3:  Right shift addition result by 1 and increment exponent by 1 if addition result has carry-over
    Step 4:  Apply RZ to addition result; overflow will be clamped to DES_MAX.
    Step 5:  Calculate flags
}
```

The following IEEE Exception Flag are supported.

- INEXACT: always set this flag
- OVERFLOW: set this flag when rounding result exceeds the DES_MAX.
- UNDERFLOW: set this flag when rounding result is denormal.

math - Extended Math Function

Function
math - Extended Math Function

Description:

The *math* instruction performs extended math function on the components in src0, or src0 and src1, and write the output to the channels of dst. The type of extended math function are based on the FC[3:0] encoding in the table below.

Function Control[3:0]	Function Description
0h	Reserved
1h	INV (reciprocal)
2h	LOG
3h	EXP
4h	SQT
5h	RSQT
6h	SIN
7h	COS
8h	Reserved

INV - Inverse

Precision:1 ULP

LOG - Logarithm

Precision: If src0 is [0.5..2], **absolute error** must be no more than 2-21. If src0 is (0..0.5) or (2..+INF], **relative error** must be no more than 2-21

Note: In ALT mode log is computed as Log_2 (abs (src0))

EXP - Exponent

Precision: <= 3ULP

SQT - Square Root

Precision: <= 1ULP

Notes: In ALT mode SQRT is computed as SQRT(abs (src0))

RSQT - Reciprocal Square Root

Precision: <= 3ULP

Notes: In ALT mode RSQ is computed as RSQ(abs (src0))

SIN - SINE

Precision: Absolute error <= 0.0008 for the range of +/- 32767* pi

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SIN is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

COS - COSINE

Precision: Absolute error <= 0.0008 for the range of +/- 32767* pi

Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of COS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.

EUISA Structures

Name	Source
AddrSubRegNum	Eulsa
EU_INSTRUCTION_BASIC_ONE_SRC	Eulsa
EU_INSTRUCTION_BASIC_THREE_SRC	Eulsa
EU_INSTRUCTION_BASIC_TWO_SRC	Eulsa
EU_INSTRUCTION_BRANCH_ONE_SRC	Eulsa
EU_INSTRUCTION_BRANCH_TWO_SRC	Eulsa
EU_INSTRUCTION_ILLEGAL	Eulsa
EU_INSTRUCTION_NOP	Eulsa

EUISA Enumerations

Name	Source
AddrMode	Eulsa
ExecSize	Eulsa
FC	Eulsa
HorzStride	Eulsa
PredCtrl	Eulsa
SFID	Eulsa
SrcMod	Eulsa
VertStride	Eulsa
Width	Eulsa

EU Programming Guide

Describes they syntax, gives examples, and provides fulsim extensions.

Fulsim Extenstions

A register can be declared as a symbol using the following form

.**declare** <symbol>**Base**=RegFile RegBase {.SubRegBase} **ElementSize**=ElementSize {**SrcRegion**=DefaultSrcRegion} {**DstRegion**=DefaultDstRegion} {**Type**=DefaultType}

The register file, the base of the register origin and the element size (in unit of bytes) are the mandatory parameters for a declared register region. Optionally, the base of the sub-register address, the default source region, the default destination region and the default type can be provided in the declaration for the symbol.

For immediate register addressing mode, the declared symbol can be used in the following Cartesian form

<symbol>(RegOff, SubRegOff)<=RegNum = *RegBase*+ **RegOff***;* SubRegNum = *SubRegBase*+ **SubRegOff**

or in the following simplified row-aligned form

<symbol>(RegOff)<=RegNum = *RegBase*+ **RegOff;** SubRegNum = *SubRegBase*

Example Declare:

.declare uwPAKMBA Base=r48 ElementSize=2 Type=uw

.declare wINTRAMB_FLAG_B Base=r8.1 ElementSize=2 Type=w

Example Use:

and (1|M0) (eq)f0.0 wINTRAMB_FLAG_B(0)<1>:w uwPAKMBA(1,0)<0;1,0>:w 0x2000:w

.default_execution_size (EXEC_SIZE)

Where executions size is :1, 4, 8, 16

Example:

.default_execution_size (16)

mov (M0) r45.0<8;8,1>:d 0:d

.default_register_type <type>

Example:

.default_register_type :d

mov (16|M0) r45.0<8;8,1> 0

Usage Examples

Examples of various instructions use.

SEND Instruction

SFID	
cre	Check and Refinement Engine
dc0	Data Cache Data Port 0
dc1	Data Cache Data Port 1
dc2	Data Cache Data Port 2

SFID	
dcr0	Data Cache Read Only Data Port
gtwy	Message Gateway
null	Null
pixi	Pixel Interpolation
rc	Render Cache Data Port
smpl	Sampler
ts	Thread Spawner
ub	URB
vme	Video Motion Estimation
cre	Check and Refinement Engine

SFID	
cre	Check and Refinement Engine
dc0	Data Cache Data Port 0
dc1	Data Cache Data Port 1
dc2	Data Cache Data Port 2
dcr0	Data Cache Read Only Data Port
gtwy	Message Gateway
null	Null
pixi	Pixel Interpolation
rc	Render Cache Data Port
ts	Thread Spawner
ub	URB
vme	Video Motion Estimation
cre	Check and Refinement Engine

send.cre (8|M0) r3 r0 r38 0x82 0x02127019	
send	Send
cre	SFID
(8|M0)	ExecSize: 8. Execution Mask Offset: 0.
r3:f	Destination. GRF Register type. Register Number: 3. Type: Float.
r0	Source 0. GRF Register type. Register Number: 0. Type: None.
r38	Source 1. GRF Register type. Register NUmber: 38. Type: None.
0x82	Extended Message Descriptor. See SEND message for more information.
0x02127019	Message Descriptor. See SEND message for more information.
(W) send.cre (8|M0) null r25 null a0.2 a0.0 {EOT}	
(W)	Write Enable is ON.
send	Send
cre	SFID
(8|M0)	ExecSize: 8. Execution Mask Offset: 0.
null	Destination. ARF NULL Register. Type: UD. This instruction does not return any values.
r25	Source 0.
null	Source 1. Can be null.
a0.2	Extended Message Descriptor.
a0.0	Message Descriptor. Indirect. Value stored in ARF address register a0.0.
EOT	End Of Thread

EU Instructions

Execution Mask	
M0	N1/Q1/H1
M4	N2
M8	N3/Q2
M12	N4
M16	N5/Q3/H2
M20	N6
M24	N7/Q4
M28	N8
(W) add (1|M0) a0.0<1>:ud r1.0<0;1,0>:ud 0x2190000:ud { @2 }	
(W)	Write Enable is ON
add	opcode add
(1|M0)	Execution size: 1. Execution Mask Offset: 0.
a0.0<1>:ud	Architecture Register File (ARF) registers: address. Register Number: 0. SubRegister Number 0.

Execution Mask	
	Destination stride: 1. Type: UD.
r1.0<0;1,0>:ud	GRF Register. Register Number: 1. SubRegister Number 0. Scalar access.
@2	Register Dependency Distance
madm (4|M0) r46.acc3:df r38.noacc:df r36.noacc:df r32.acc2:df Write Enable is OFF	
madm	opcode madm
(4|M0)	Execution size: 4. Execution Mask Offset: 0.
r46.acc3:df	GRF register. Register Number: 46. SubRegister Number: 0. ACC3 is updated to accumulate precision. See MADM for more information. Type: DF.
r38.noacc:df	GRF register. Register Number: 38. SubRegister Number: 0. No ACC register is updated. See Accumulator register for more information. Type DF.
r32.acc2:df	GRF register. Register Number: 32. SubRegister Number: 0. ACC2 is used for extra precision.
math.invm (4|M4) (eo)f0.0 r8.acc2:df r36.noacc:df r34.noacc:df Write enable is OFF	
math.invm	opcode MATH, Extended Math Function INVM (Inverse with higher precision).
(4|M0)	Execution Size: 4. Execution Mask Offset: 4:
(eo)f0.0	Flag register as implicit destination, with a flag modifier. See math instruction for more information.
cmp (1|M0) (lt)f0.0 null.0<1>:d r35.0<0;1,0>:d r1.7<0;1,0>:d Write enable is OFF	
cmp	Opcode compare
(1|M0)	Execution Size: 1. Execution Mask Offset: 0.
(lt)f0.0	Flag register as implicit destination, with Less Then flag modifier. See cmp instruction for more information.
r35.0<0;1,0>:d	GRF register type. Register Number: 35. SubRegister Number: 0. Region: <0;1,0>, Scalar. vStride: 0. Width: 1. hStride: 0.
sel (1|M0) (lt)f0.0 r2.4<1>:d r2.4<0;1,0>:d r1.2<0;1,0>:d Write enable is OFF	
sel	Opcode select.
(1|M0)	Execution Size: 1. Execution Mask Offset: 0.
(lt)f0.0	Flag register as implicit destination, with Lest Then flag modifier. See sel instruction for more information.
(W&~f0.0) sel (8|M0) r2.0<1>:f r2.0<8;8,1>:f 0xBF800000:f	
(W&~f0.0)	Write Enable is ON. This instruction is predicated. Flag register f0.0 is used, and it's values are negated. See select instruction for more information.
mad (8|M0) r15.0<1>:f r14.0<0;0>:f r3.0<8;1>:f r10.0<0>:f { $2 } Write enable is OFF	
mad	Opcode: mad.
(8|M0)	Execution Size: 8. Execution Mask Offset: 0.
r15.0<1>:f	Destination. Horizontal Stride: 1. Type: Float.

Execution Mask	
r14.0<0;0>:f	Src0. vStride: 0. hStride: 0. Scalar.
r3.0<8;1>:f	Src1. vStride: 1. hStride: 1. Vector.
r10.0<0>:f	Src2. hStride: 0. Scalar
$2	one of the source register is dependent on a send instruction using token $2
mad (8|M0) r15.0<1>:f r14.0<0;0>:f r3.0<8;1>:f r10.0<1>:f	
r10.0<1>:f	Src2. hStride: 1. Vector.

SEND Instructions

Sends (8|M0) r3:f r0 r38 0x82 0x02127019	
Sends	Split Send. Two Sources.
(8|M0)	ExecSize: 8. Execution Mask Offset: 0.
r3:f	Destination. GRF Register type. Register Number: 3. Type: Float.
r0	Source 0. GRF Register type. Register Number: 0. Type: None.
r38	Source 1. GRF Register type. Register Number: 38. Type: None.
0x82	Extended Message Descriptor. See SEND message for more information.
0x02127019	Message Descriptor. See SEND message for more information.
Send (8|M0) r0:f r37:d 0x2 0x08127019	
Send	Send instruction. One Source.
(8|M0)	ExecSize: 8. Execution Mask Offset: 0.
r0:f	Destination. GRF Register Type. Register Number: 0. Type: Float.
r37:d	Source 0. GRF Register Type. Register Number: 0. Type: D.
0x2	Message Descriptor.
0x08127019	Extended Message Descriptor.
Send (16|M0) r3:w r23:ud 0x2 a0.0	
Send	Send instruction. One source.
(16|M0)	ExecSize: 16. Execution Mask Offset: 0.
r3:w	Destination.
r23:ud	Source 0.
0x2	Extended Message Descriptor
a0.0	Message Descriptor. Indirect. Value stored in ARF address register a0.0.
(W) sends (8|M0) null:ud r25 r19 0x10c a0.0	
(W)	Write Enable is ON.
sends	Split Send.
(8|M0)	ExecSize: 8. Execution Mask Offset: 0.
null:ud	Destination. ARF NULL Register. Type: UD. This instruction does not return any values.
r25	Source 0.

Sends (8|M0) r3:f r0 r38 0x82 0x02127019	
r19	Source 1.
0x10C	Extended Message Descriptor.
a0.0	Message Descriptor. Indirect. Value stored in ARF address register a0.0.

Control Flow Instructions

(W) jmpi (1|M0) L1320	
(W)	Write Enable is ON.
jmpi.	Jump Immediate.
(1|M0)	Execution Size: 1. Execution Mask Offset: 0.
L130	Label to jump to.
(W&~f0.1) jmpi (1|M0) L904	
(W&~f0.1)	Write Enable is ON. ARF flag register used as Predicate. SubRegister: 1. Values are inverted.
(~f0.0) if (4|M0) L4744 L4760 // (4|M0) instructions L4744: // (4|M0) instructions L4760: endif (4|M0) L4776 L4776:	
(~f0.0)	ARF Flag register used as a predicate. SubRegister: 0. Values are inverted.
if	IF
(4|M0)	Execution Size: 4. Execution Mask Offset: 0.
L4744	JIP
L4760	UIP
L4776	JIP

Shared Functions

Shared Functions are hardware units that perform operations on behalf a thread running in an EU. Shared Functions are sent a message payload by the thread, and optionally return results to the thread.

The most common shared function operations are memory surface read and write messages performed by the Sampler, the Pixel Port, and the Data Port. The following sections describe each of the shared function operations in detail.

The most common shared function operations are memory surface read and write messages performed by the Pixel Port and the Data Port. The following sections describe each of the shared function operations in detail.

Shared functions are identified by their Shared Function ID (SFID), which is one of the parameters to the EU send instruction.

List of Shared Function Identifiers

Programming Note	
Context:	Shared Function ID
SFID_DP_DC1 is an extension of SFID_DP_DC0 to allow for more message types. They act as a single logical entity.	

Programming Note	
Context:	Shared Function ID
SFID_DP_DC1 , SFID_DP_DC2, and SFID_P_DC3 are extensions of SFID_DP_DC0 to allow for more messages types. They act as a single logical entity.	

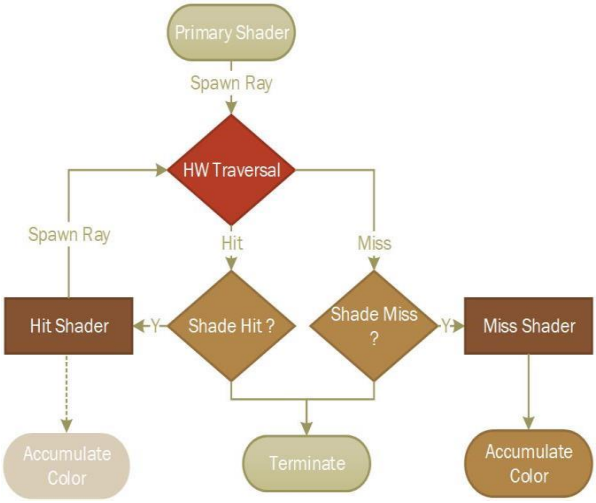
Ray Tracing

This section describes new shared function: Ray Tracing. Ray Tracing Hardware Block is responsible for processing and accelerating TraceRay message from a shader by performing HW traversal of acceleration structures (aka BVH - Binding Volume Hierarchy). TraceRay message invokes HW traversal for the valid rays in the SIMD8/16 messages.

Primary Rays are generated via DispatchRay command on compute pipeline. Each primary ray gets a unique ID (StackID) allocated by the thread dispatcher. This StackID is used by both primary ray shader as well as all continuation shaders and HW traversal function to store various data structures in memory. These data structures include: Ray, Call Stack, TraversalStack, Potential Hit Record, Committed Hit Record etc. Memory layout of these datastructures are described in the subsequent section of the Bspec. Any shader (including primary ray shader) generating a TraceRay message, saves the live register and terminates after storing the necessary data for the sub-sequent processing in HW or by the shader. This guarantees that only one of the thread can access data at any given StackID. Further, there is a clear synchronization required between HW and Shader to manage producer-consumer model on data in memory for a given StackID.

Basic Flow For Ray Tracing

Pictorially, the basic Ray Tracing flow is described in this diagram:



Primary Shader initializes the top level ray and committedHitInfo and sends TraceRayMessage with `traceRayCtrl = TRACE_RAY_INITIAL`. The RTUnit executes the following steps:

- Compute ray address, read top level ray and compute ray setup
- Begin traversal of the top level BVH
- On hitting an instance leaf
    - If previously committed hit
        - Write hit to commitedHitInfo
    - If the traversal stack has entries
        - write Traversal Stack
    - If HW Instancing mode
        - Transform ray and compute ray setup
        - goto traversal of the bottom level BVH
    - Else if SW Instancing mode
        - Write instance information to potentialHitInfo
        - Launch an instance shader
- If no more hits
    - Send a miss shader request to the BTD

Instance Shader initializes the bottom level ray and sends TraceRayMessage with `traceRayCtrl = TRACE_RAY_INSTANCE`. The RTUnit executes the following steps:

- Compute ray address, read ray and compute ray setup
- Read committedHitInfo and update previously committed
- Begin traversal of the bottom level BVH

- On hitting a quad leaf
    - If any-hit shader is needed (see the DirectX Ray Tracing API spec)
        - If shader is null
            - Continue traversal
        - If HW instancing mode and ray not updated
            - Write the transformed ray and mark as updated
        - Write current uncommitted hit to potentialHitInfo
        - If previously committed hit
            - Write hit to commitedHitInfo
        - If the traversal stack has entries
            - Write traversal stack
        - Send an any-hit shader request to the BTD
    - else
        - update previously committed hit
- On hitting a procedural leaf
    - If shader is null
        - Continue traversal
    - If HW instancing mode and ray not updated
        - Write the transformed ray and mark as updated
    - Write information for current leaf to potentialHitInfo
    - If previously committed hit
        - Write hit to commitedHitInfo
    - If the traversal stack has entries
        - Write traversal stack
    - Send an intersection shader request to the BTD
- If no more hits
    - Decrement bvhLevel
    - Read traversal stack
    - If traversal stack is empty
        - Send miss shader request to the BTD
    - else
        - goto read top level ray

The Any-Hit shader sends a TraceRayMessage with `traceRayCtrl = TRACE_RAY_COMMIT` if the potential Hit is committed (see the DirectX Ray Tracing API spec). The RTUnit executes the following steps:

- Compute ray address and read ray
- Read potentialHitInfo and update previous committed hit

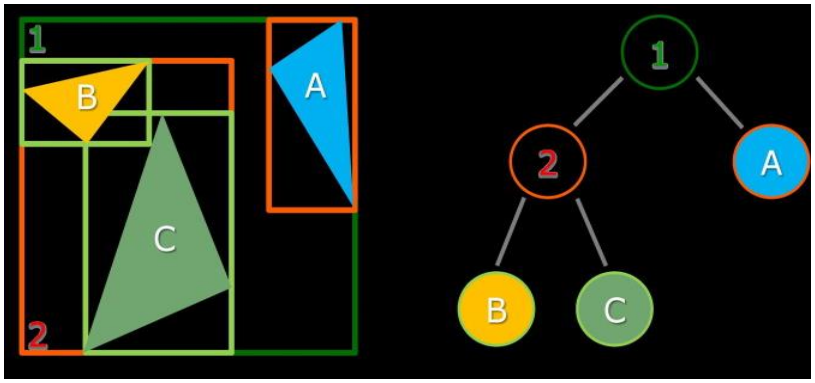
- goto traversal of the bottom level BVH

Any-Hit Shader sends TraceRayMessage with `traceRayCtrl = TRACE_RAY_CONTINUE` if the potential Hit is not committed. The RTUnit executes the following steps:

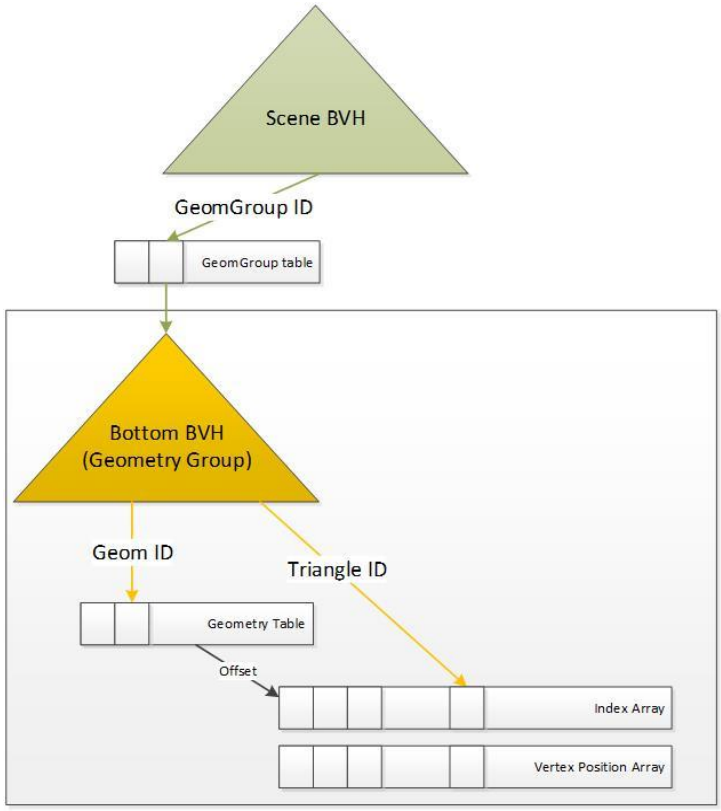
- Compute ray address and read ray
- Read commitedHitInfo and update previous committed hit
- goto traversal of the bottom level BVH

Ray Tracing Acceleration Structure : BVH

Ray Tracing requires acceleration structure for ray traversal. This acceleration structure is called Binding Volume Hierarchy (BVH). BVH is created based on geometry information a priori before binding and submitting the DispatchRay, a GPGPU walker command. Current BVH is based on quantization technique with up to 2 children for every node (except leaf node), therefore it is also referred to as QBVH2. Following diagram shows a very simple view of a BVH encapsulating the simple geometry. Here, the color codes of the node depicts the associated bounding boxes and leaf primitives.



BVH captures geometry using multiple levels as shown in the picture below:



Memory Based Data Structures For Ray Tracing

Any DIspatchRay Command requires allocation of per Ray storage in the graphics memory. Per-ray storage is accessed by the unique id. This unique ID is logically described with the hierarchy of namespaces: CSLICE_ID.DSSID.StackID. CSLICE_ID refers to the compute slice in the entire machine, DSSID refers to the DSS within the compute slice and StackID refers to the StackID assigned within DSS hardware to a primary Ray. The linear per-ray storage in the memory holds multiple data-structures as described below:

Per-Ray RTStack data structure consists of these data-structures:

{ TopRay(ray for the top level BVH), TopTravesrsalStack (traversal stack for top level BVH), BottomRay (ray for the bottom level BVH), BottomTraversalStack(traversal stack for the bottom level BVH), CommittedHitInfo, PotentialHitInfo, ScratchSpace }.

Logical layout of the RTStack is linear array of the fields (from left to write as described above - mapped from LSB to MSB) at the address computed by

RTSatckAddress(RTStackBase, StackID, RTStackSize) = RTSatckBase + CSLICE_ID*CSLICE_STACK_SIZE + DSSID*DSS_STACK_SIZE + StackID*RTStackSize;

Where, CSLICE_STACK_SIZE = (Number of DSS in the compute slice)*DSS_STACK_SIZE;

DSS_STACK_SIZE = (Number of StackIDs per DSS) * RTStackSIze;

Exact details of the data-structures are described below:

The RTStack is a ray tracing stack per ray/SIMD lane. It is used to communicate ray and hit information between EU and ray tracing hardware, and to store internal hardware state to memory. The stack contains a first part that is read and written by hardware and software, while the second part is completely software defined. Software can use this space to spill life registers over ray tracing calls.

RTStack

(bits are mapped from LSB to MSB of a dword. dwords are listed by increasing address)

Field	Width(bits)	Description
committedHit	MemHit	stores committed hit
potentialHit	MemHit	stores potential hit that is passed to any hit shader

Memory Hit Structure: This hit structure is present in memory inside the ray tracing stack. Each time the ray tracing hardware transitions to the EU, its internal hit structure (see below) may have to get converted to a MemHit and written out to main memory.

MemHit

(bits are mapped from LSB to MSB of a dword. dwords are listed by increasing address)

Field	Width(bits)	Description
t	32	hit distance of current hit (or initial traversal distance)
u	32	barycentric hit coordinates
v	32	barycentric hit coordinates
primIndexDelta	16	prim index delta for compressed meshlets and quads
valid	1	set if there is a hit
leafType	3	type of node primLeafPtr is pointing to
primLeafIndex	4	index of the hit primitive inside the leaf
bvhLevel	3	the instancing level at which the hit occurred
frontFace	1	whether we hit the front-facing side of a triangle (also used to pass opaque flag when calling intersection shaders)
pad0	4	unused bits
primLeafPtr	42	pointer to BVH leaf node (multiple of 64 bytes)
hitGroupRecPtr0	22	LSB of hit group record of the hit triangle (multiple of 16 bytes)
instLeafPtr	42	pointer to BVH instance leaf node (in multiple of 64 bytes)
hitGroupRecPtr1	22	MSB of hit group record of the hit triangle (multiple of 32 bytes)

Memory Ray Structure: This ray structure is present in memory inside the ray tracing stack. Each time the ray tracing hardware transitions to the EU, its internal ray structure(see below) may have to get converted to a MemRay and written out to main memory.

MemRay

Programming Note
bits are mapped from LSB to MSB of a dword. dwords are listed by increasing address)

Field	Width(bits)	Description
org_x	32	the origin of the ray
org_y	32	the origin of the ray
org_z	32	the origin of the ray
dir_x	32	the direction of the ray
dir_y	32	the direction of the ray
dir_z	32	the direction of the ray
tnear	32	the start of the ray
tfar	32	the end of the ray
rootNodePtr	48	root node to start traversal at
rayFlags	16	ray flags (see RayFlag structure)
hitGroupSRBasePtr	48	base of hit group shader record array (8-bytes alignment)
hitGroupSRStride	16	stride of hit group shader record array (8-bytes alignment)
missSRPtr	48	pointer to miss shader record to invoke on a miss (8-bytes alignment)
pad	8	
shaderIndexMultiplier	8	shader index multiplier
instLeafPtr	48	the pointer to instance leaf in case we traverse an instance (64-bytes alignment)
rayMask	8	ray mask used for ray masking

Traversal Stack (TopTraversalStack and BottomTraversalStack are described using this data structure)

This data-structure is written and read by RT Shared Function. There are up to 5 such TraversalStackEntries per depth are allocated.

(Described from LSB to MSB)

Field	Width(bits)	Description
restartTrail0	29	Restart trail for each level
restartTrail1	29	Restart trail for each level
nextLeafPrimitiveIndex	3	Index to the next primitive in the leaf
restartDepth	5	number of levels of BVH are in the restart depth
stackDepth	25	depth for each inner stack entry
nodeTypeMSB	5	node type
StackEntry0.Offset	30	multiple of 64B, max prims = 2^29 and max nodes = 2^30
nodeTypeLSB	2	

Field	Width(bits)	Description
StackEntry1.Offset	30	multiple of 64B, max prims = 2^29 and max nodes = 2^30
nodeTypeLSB	2	
StackEntry2.Offset	30	multiple of 64B, max prims = 2^29 and max nodes = 2^30
nodeTypeLSB	2	
StackEntry3.Offset	30	multiple of 64B, max prims = 2^29 and max nodes = 2^30
nodeTypeLSB	2	
StackEntry4.Offset	30	multiple of 64B, max prims = 2^29 and max nodes = 2^30
nodeTypeLSB	2	

RayFlags Enumeration Structure (as referred to in MemRay by rayFlags field)

Name	Value	Description
NONE	0x0000	No flags are set
FORCE_OPAQUE	0x0001	Forces geometry to be opaque
FORCE_NON_OPAQUE	0x0002	Forces geometry to be non-opaque
ACCEPT_FIRST_HIT_AND_END_SEARCH	0x0004	Terminates traversal on the first hit found (typically used for shadow rays)
SKIP_CLOSEST_HIT_SHADER	0x0008	Skip the execution of the closest hit shader
CULL_BACK_FACING_TRIANGLES	0x0010	Back face triangles do not produce hit
CULL_FRONT_FACING_TRIANGLES	0x0020	Front face triangles do not produce hit
CULL_OPAQUE	0x0040	Opaque geometry does not produce hit
CULL_NON_OPAQUE	0x0080	Non-opaque geometry does not produce hit
SKIP_TRIANGLES	0x0100	Skip all triangle intersections and consider them as misses
SKIP_PROCEDURAL_PRIMITIVES	0x0200	Skip execution of intersection shaders for procedural primitives
TRIANGLE_FRONT_COUNTERCLOCKWISE	0x4000	This value MUST not be programmed by SW but used internally by HW only. This is added to this table to avoid future collision with this value from API. Therefore, SW must not set this bit in the bit-array.
LEVEL_ASCEND_DISABLED	0x8000	Disables the automatic ascend from level n BVH to level (n-1) BVH when traversal is complete at level n

State Model for Ray Tracing

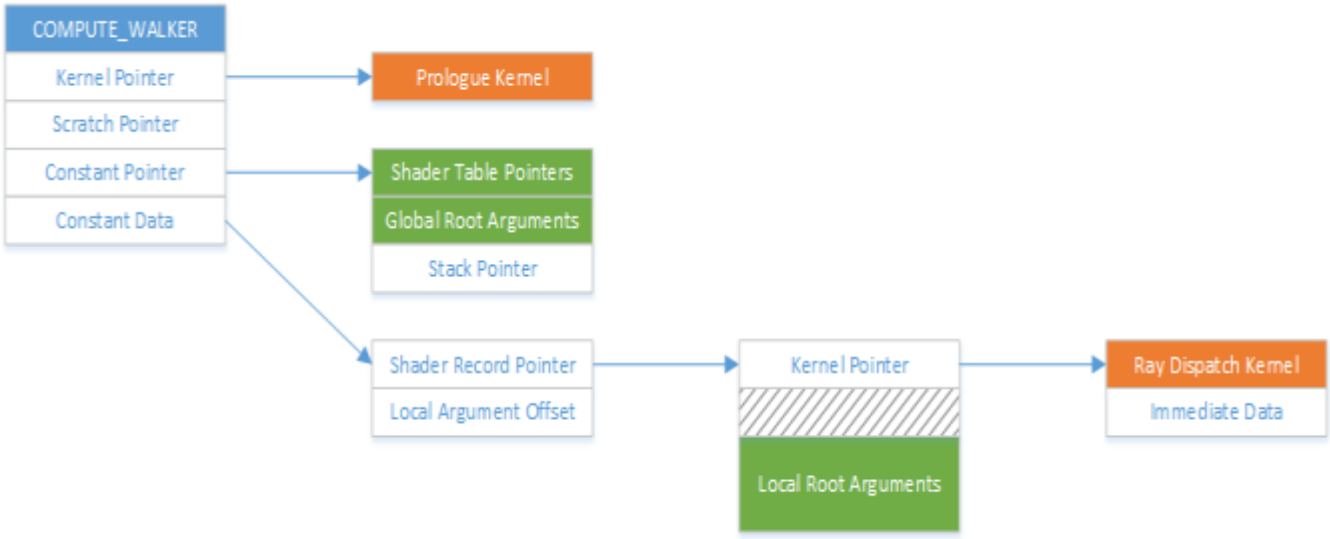
RT shared function requires certain set of global constants in order to address various sections in the graphics virtual memory. These global constants are at the Global Constants pointer generated by the TraceRay (or Spawn Message) message after the prolog shader follows the constant pointer in the DispatchRay command. RT shared function fetches these global constants and pipelines the Global Arguments pointer with each bindless dispatch caused by the TraceRay within the scope of the DispatchRay command. The following fields are programmed from LSB to MSB.

RTDispatchGlobals

(bits are mapped from LSB to MSB of a dword. dwords are listed by increasing address)

Field	Width(bits)	Description
rtMemBasePtr	64	[47:6] 64B aligned GPUVA points to the base of the RTStack for all StackIDs for the entire GPU.
callStackHandlerPtr	64	This is a continuation handler KSP that is invoked by BTD when shader KSP is NULL. It has the same format has KSP i.e. [63:32] MBZ. callStackHandlePtr[4] indicates Bindless Thread dispatch mode and should be programmed in the same way as it is in programmed in the Bindless Shader Record.
stackSizePerRay	32	maximal stack size of a ray [31:8]: MBZ [7:0]: indicates a multiplier in terms of 64B to compute the stackSizePerRay
numDSSRTStacks	32	[31:12] : MBZ [11:0]: number of StackIDs Defines the maximum number of StackIDs per DSS for async ray tracing. This field must not exceed 2K. It can be programmed lower per computer walker command to lower the outstanding StackIDs in the DSS. For Sync Ray tracing (i.e. using RayQueries), SW must allocate space assuming 2K StackIDs. Setting this field 0 is not allowed.
maxBVHLevels	32	[31:3] : MBZ [2:0] MaxBVHLevels HW interprets [2:0] = 0 as 8 and values of 1 through 7 are interpreted as is.
leqTestEnable	1	When this bit is set, Ray-Triangle test uses LEQ as comparison instead of LE.
perContextHWBits	31	MBZ
hitGroupBasePtr	64	[63:48] MBZ [47:3] 8B aligned GPUVA : base pointer of hit group shader record array (8-bytes alignment) [2:0] MBZ
missShaderBasePtr	64	[63:48] MBZ [47:3] 8B aligned GPUVA: base pointer of miss shader record array (8-bytes

Field	Width(bits)	Description
		alignment) [2:0] MBZ
callableShaderBasePtr	64	[63:48] MBZ [47:3] 8B aligned GPUVA: base pointer of callable shader record array (8-bytes alignment) [2:0] MBZ
hitGroupStride	32	stride of hit group shader records (8-bytes alignment)
missShaderStride	32	stride of miss shader records (8-bytes alignment)
callableShaderStride	32	stride of callable shader records (8-bytes alignment)



Messages to Ray Tracing Function

Primary Ray shaders or other HitGroup/Miss shaders send the TraceRay messages to Ray Tracing Shared Function (RT Shared Function). Trace Ray message has a SIMD8/SIMD16 versions with great similarity to BTD spawn message, but an additional field TraceRayControl determines the behavior of each valid ray (i.e. SIMD lane in the SIMD message) in the RT shared function. This TraceRayControl enumeration is as follows:

TraceRayControl field Enumeration

Value	Encoding	Description
TRACE_RAY_INITAL	0	RT shared function loads memRay, ignores the hit info and traversal stack . Typically used by the primary ray shader.
TRACE_RAY_INSTANCE	1	RT shared function begins traversing within the instance i.e., in bottom level acceleration structure. It loads memRay and CommittedHitInfo.
TRACE_RAY_COMMIT	2	RT shared function loads memRay, PotentialHitInfo and traversal stack. It updates to CommittedHitInfo and continues traversal.

Value	Encoding	Description
TRACE_RAY_CONTINUE	3	RT shared function loads memRay, CommittedHitInfo and traversal stack. It continues traversal.

Ray Tracing Address Computation for Memory Resident Structures

There are several address computations that are important for accessing the various structures in the memory. These address computations are described in the equations below with previously defined terms.

For asynchronous ray tracing, stackID for each ray is assigned by HW and it is present in the TraceRay message for each SIMD lane. For Synchronous ray tracing (i.e., when RayQuery bit is set in the TraceRay message), stackID is computed as follows by HW:

Programming Note	
Context:	StackID Computation for RayQuery

SyncStackID = (EUID[3:0] « 7) | (ThreadID[2:0] « 4) | SIMDLaneID[3:0]; // With fused EUs
SyncStackID = (EUID[2:0] « 8) | (ThreadID[3:0] « 4) | SIMDLaneID[3:0]; // With natively wide EUs	

For synchronous ray tracing, HW allocates the Ray and Traversal Stack etc counting down from rtMemBasePtr.

All addresses described in this section are byte addresses.

Address Computation for Memory Based Data Structures: Ray and TraversalStack (Sync Ray Tracing)

These equations compute the byte addresses. RT shared function derives the address of the Ray and the Traversal Stack based on the BVH level.

The size of RTStack for synchronous ray tracing is computed as follows:

```
     syncStackSize = if (maxBVHLevels %2 == 1)? (sizeof(HitInfo)*2 + (sizeof(Ray) +
sizeof(TravStack))* maxBVHLevels + 32B) :
                                                (sizeof(HitInfo)*2 + (sizeof(Ray) +
sizeof(TravStack))* maxBVHLevels);
     syncBase = RTDispatchGlobals.rtMemBasePtr – (DSSID * NUM_SIMD_LANES_PER_DSS + SyncStackID
+ 1)*syncStackSize;
     rayBase = syncBase + sizeof(HitInfo) * 2;
     rayPtr  = rayBase + bvhLevel * sizeof(Ray);
     travStackPtr = rayBase + RTDispatchGlobals.maxBVHLevels * sizeof(Ray) + bvhLevel *
sizeof(TravStack);
```

Address Computation for Memory Based Data Structures: Ray and TraversalStack (Async Ray Tracing)

These equations compute the byte addresses. RT shared function derives the address of the Ray and the Traversal Stack based on the BVH level.

```
    stackBase = RTDispatchGlobals.rtMemBasePtr + (DSSID * RTDispatchGlobals.numDSSRTStacks +
stackID) * RTDispatchGlobals.stackSizePerRay // 64B aligned
    rayBase =  stackBase + sizeof(HitInfo) * 2 // 64B aligned
```

```
    rayPtr   = rayBase + bvhLevel * sizeof(Ray); // 64B aligned
    travStackPtr = maxBVHLevels * sizeof(Ray) + bvhLevel * sizeof(TravStack); //32B aligned
```

Here, sizeof(Ray) = 64B, sizeof(HitInfo) = 32B, sizeof(TravStack) = 32B.

Address Computation of Shadder Record Pointer for Software based instancing

The base address of the instance shader record array is computed in the primary shader and stored in the top level ray structure.

```
 ray.hitGroupOrInstSRBasePtr = PatchedinISA();
 ray.SRStride = PatchedinISA(); // typically an 8 byte stride
```

The RT shared function computes the pointer to the instance shader record based on the instShaderRecordID in the leaf node of the top level BVH.

```
 instShaderRecordPtr = ray.hitGroupOrInstSRBasePtr + InstanceLeaf.instShaderRecordID *
ray.SRStride;
```

The base address of the hit shader record is computed in the instance shader and stored in the bottom level ray structure.

```
 ray.hitGroupOrInstSRBasePtr = RTDispatchGlobals.hitGroupBasePtr +
RTDispatchGlobals.hitGroupStride * (RTState.rayHitGroupOffset  + InstanceLeaf.HitGroupOffset)
 ray.SRStride = RTDispatchGlobals.hitGroupStride *
MultiplierForGeometryContributionToHitGroupIndex; // from TraceRay
```

The RT sharesd function computes the pointer to the hit group shader record based on the geomID value stored in the leaf node. A hit group shader record includes two shader records, one for the closest hit shader and one for a combined any hit shader and intersection shader.

```
 hitGrpShaderRecordPtr = ray.hitGroupOrInstSRBasePtr + QuadLeaf.geomIndex * ray.SRStride;
 anyHitShaderRecordPtr = hitGrpShaderRecordPtr; // intersection shader is at this offset too
 closestHitShaderRecordPtr = hitGrpShaderRecordPtr + 8;
```

Address Computation for Shader Record Pointer for HW based instancing

With HW instancing, the primary shader writes the base address, stride and geom multiplier for computing the hitGroup shader record pointer in the top level ray.

```
 ray.hitGroupOrInstSRBasePtr = RTDispatchGlobals.hitGroupBasePtr +
RTDispatchGlobals.hitGroupStride * RTState.rayHitGroupOffset;
 ray.SRStride = RTDispatchGlobals.hitGroupStride;
 ray.hitGroupGeomMul = MultiplierForGeometryContributionToHitGroupIndex;
```

On hitting an instance the RTUnit updates its internal storage for the hitGroupOrInstSRBasePtr.

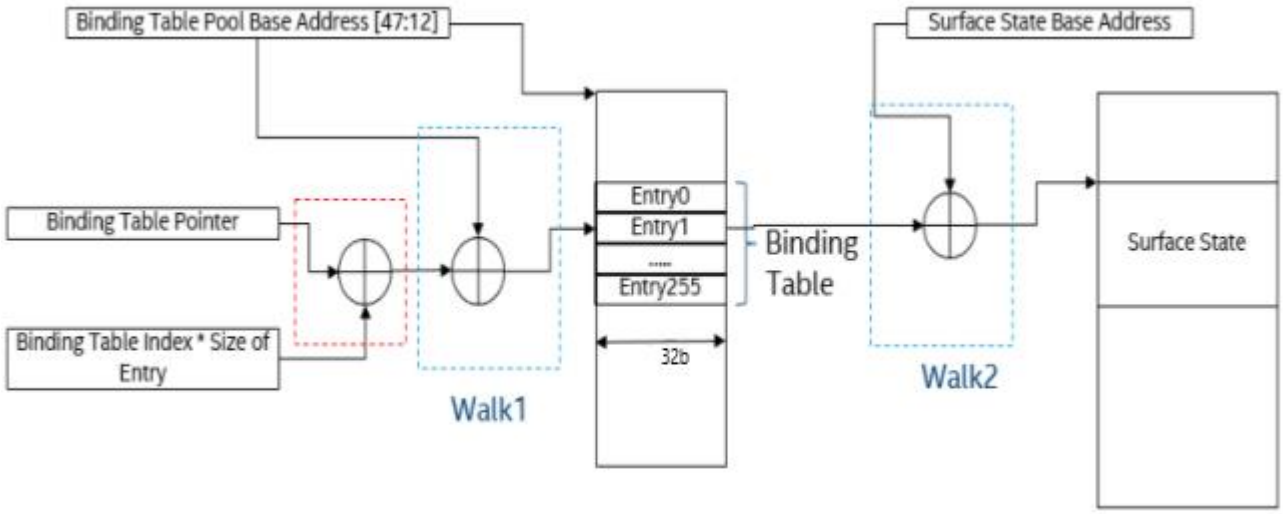
```
 hitGroupOrInstSRBasePtr = hitGroupOrInstSRBasePtr + ray.SRStride *
InstanceLeaf.HitGroupOffset;
```

On hitting a primitive leaf in the bottom level BVH, the RTUnit computes the pointer to the hit group shader record based on the geomID value stored in the leaf node. A hit group shader record includes two shader records, one for the closest hit shader and one for a combined any hit shader and intersection shader.

```
 hitGrpShaderRecordPtr = hitGroupOrInstSRBasePtr + QuadLeaf.geomIndex * ray.hitGroupGeomMul *
ray.SRStride;
 anyHitShaderRecordPtr = hitGrpShaderRecordPtr; // intersection shader is at this offset too
 closestHitShaderRecordPtr = hitGrpShaderRecordPtr + 8;
```

Binding Table

The surface state model is intended to be used for constant buffers, render target surfaces, and media surfaces.



Below is a table specifying the different configurations of the Binding Table Pointers and the BINDING_TABLE_STATE. This table applies for render, position, and compute engines. Each engine supports a separate address.

Binding Table Entry Format	Binding Table Pointer	Base Address
32 bit size/[31:6]	20:5	Binding Table Pool Address

The data port uses the binding table to retrieve surface state when the Binding Table Index is less than 240. See the following for a definition of the binding table:

- **SW Generated BINDING_TABLE_STATE**

Fused Send Message Handling

Shared Functions receive messages from the EU. The EU kernel executes a SEND instruction to transmit the message payload to the Shared Function. The message payload contains a Message Descriptor, an optional Message Header, and a message-specific Address and Data Payload.

The EU can mark two SEND messages to be combined together as a single operation.

Combine Type	Behavioral Description and Restrictions
Atomic Send Message	A single thread issues back-to-back SEND messages. The two messages are sent as an indivisible pair to the shared function. The shared function processes the pair of messages sequentially. There is no guarantee that the two messages are executed back-to-back by the shared function: another

Combine Type	Behavioral Description and Restrictions
	EU's send message may execute between the two messages.
Fused Send Message	A fused-EU pair of threads issues a SEND message. The two messages are sent as an indivisible pair to the shared function. The full extended message descriptor and the Message Header payload are required to be the same for the paired messages. Only the Address and Data payload are independent for the paired messages. The shared function processes the paired messages simultaneously.
Atomic Fused Send Message	When combined with an Atomic Send, the "atomic chained" messages from an EU is sent first, before the messages from other EU of the fused-pair is sent. Atomic Send is only allowed for URB writes messages, and a maximum of 2 such send messages can be chained.

3D Sampler

The 3D Sampling Engine provides the capability of advanced sampling and filtering of surfaces in memory.

The sampling engine function is responsible for providing filtered texture values to the EU in response to sampling engine messages. The sampling engine uses SAMPLER_STATE to control filtering modes, address control modes, and other features of the sampling engine. A pointer to the sampler state is delivered with each message, and an index selects one of 16 states pointed to by the pointer. Some messages do not require SAMPLER_STATE. In addition, the sampling engine uses SURFACE_STATE to define the attributes of the surface being sampled. This includes the location, size, and format of the surface as well as other attributes.

Although data is commonly used for "texturing" of 3D surfaces, the data can be used for any purpose once returned to the execution core. The 3D Sampler can be used to assist the media sampler in specific operations such as video scaling.

The following table summarizes the various subfunctions provided by the Sampling Engine. After the appropriate subfunctions are complete, the 4-component (reduced to fewer components in some cases) filtered texture value is provided to the EU to complete the *sample* instruction.

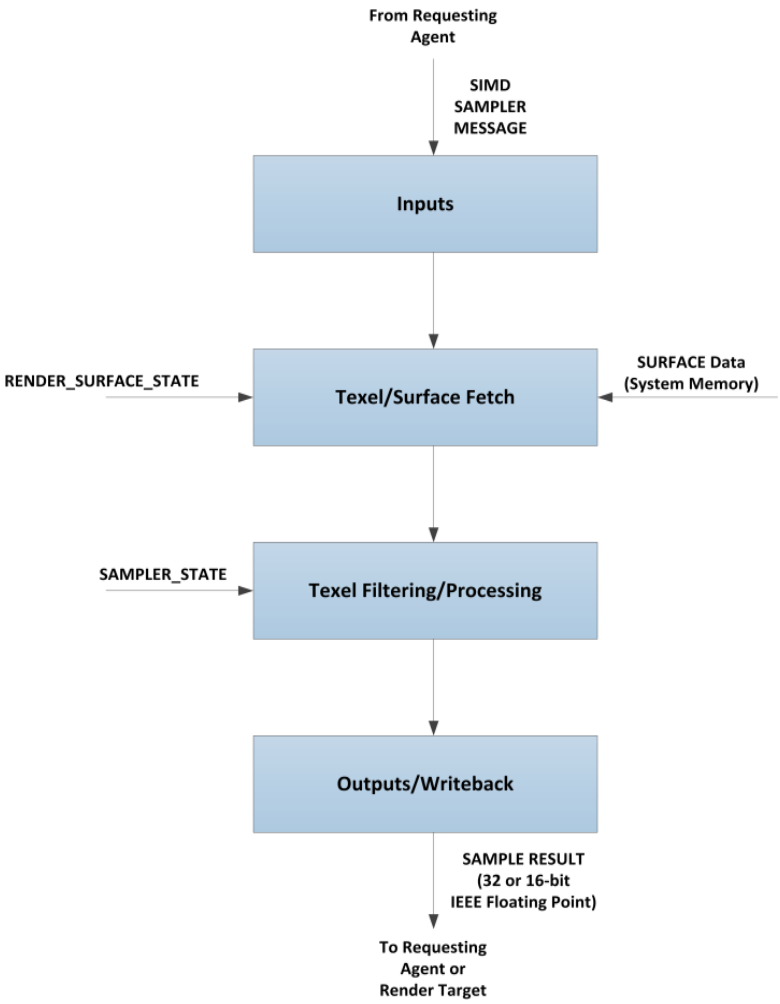
Subfunction	Description
Texture Coordinate Processing	Any required operations are performed on the incoming pixel's interpolated internal texture coordinates. These operations may include cube map intersection.
Texel Address Generation	The Sampling Engine determines the required set of texel samples (specific texel values from specific texture maps), as defined by the texture map parameters and filtering modes. This includes coordinate wrap/clamp/mirror control, mipmap LOD computation and sample and/or miplevel weighting factors to be used in the subsequent filtering operations.
Texel Fetch	The required texel samples are read from the texture map. This step may require decompression of texel data. The texel sample data is converted to an internal format.
Shadow Pre-Filter Compare	For shadow mapping, the texel samples are first compared to the 3rd (R) component of the pixel's texture coordinate. The boolean results are used in the texture filter.

Subfunction	Description
Texel Filtering	Texel samples are combined using the filter weight coefficients computed in the Texture Address Generation function. This "combination" ranges from simply passing through a "nearest" sample to blending the results of anisotropic filters performed on two mipmap levels. The output of this function is a single 4-component texel value.
Texel Color Gamma Linearization	Performs optional gamma decorrection on texel RGB (not A) values.
8x8 Video Scaler	Performs scaling using an 8x8 filter.

Sampling Engine

3D Sampler Theory of Operation

The 3D sampler (sometimes referred to as texture sampler) is a self-contained functional block within the Graphics Core which receives messages from other agents in the Graphics Core, fetches data from external memory sources typically referred to as "surfaces", performs operations on the data and returns the results in standard formats to the requester (or directly to a Render Target is requested). One of the most common applications of the 3D sampler is to return a filtered/blended pixel from a location in a texture map.



Sampler Inputs Messages

Input requests to the 3D Sampler are in the form of messages (see Messages sub-section for a description of message types and formats). A pixel shader kernal executing on the Graphics Core is an example of an agent which is capable of sending sample messages to the 3D Sampler.

In its most basic form, the sampler receives coordinates to a location within a field of data (often a texture map or depth map) and returns a value which represents the pixel color or depth which may be filtered/blended as defined by associated surface and sampler state objects. Sampler can also work on un-typed data structures called buffers.

Messages are sent in SIMD (Single Instruction Multiple Data) format where there are 8, 16, 32 or 64 coordinate tuples to be processed (i.e. SIMD8, SIMD16 etc.) in the same manner. Some message types are restricted to SIMD8 and SIMD16 varieties and other are restricted to SIMD32 or SIMD64. See the section on Texture Coordinate Processing more details on texture coordinate requirements.

SIMD8 and SIMD16 messages are further organized into groups of 4 sets of coordinates which generally form a 2x2 "subspan" of texel locations. The spatial locality of the texel locations within a sub-span improves the performance of the sampler and allows the processing of the 4 texel locations in parallel. A SIMD8 message contains two subspans and a SIMD16 contains 4 subspans.

Sampler Data Fetches

The 3D sampler will automatically fetch required data from surfaces in system memory as needed to perform each sample operation. Fetched data may be stored in an internal cache to reduce latency for subsequent fetch operations.

The sampler calculates the address into a surface and uses RENDER_SURFACE_STATE state objects to determine the location within system memory and the format of the surface being fetched. Sampler can also receive or calculate the LOD (Level of Detail) of a surface if the surface supports multiple Mips and will fetch from the correct Mip in this case. See Texture Address Calculation sub-section for more detail on addresses and LOD calculation.

The sampler will also automatically decompress any supported compression format once data has been fetched. See the subsection Surface State for a list of supported surface formats, including compressed formats. Likewise, the sampler can linearize (inverse Gamma) sRGB formats prior to filtering.

Sampler Filtering and Processing

The sampler is capable of performing all basic filtering operations (point, bilinear, trilinear, anisotropic, cube etc.) based on the SAMPLER_STATE state object associated with the sample operation being requested.

In most cases, data returned is in the form of 32-bit or 16-bit IEEE floating-point per channel to ensure maximum precision. See Writeback Message section for a description of the format of returned data. Output Data is only returned to the requesting agent or written to a designated Render Target (RT). Sample results are never cached within the sampler or written to system memory.

Sampler Caching Policies

Sampler State and Texture Caches

The 3D sampler caches both texture and state to improve performance. These read-only caches are referred to as the L1 Texture cache and L1 State cache. The L1 State cache

contains both RENDER_SURFACE_STATE and SAMPLER_STATE entries. The L1 Texture cache contains data from any surface which was sampled. It is assumed that this state and texture data, when fetched, is also cached in the L3 cache for performance reasons.

Cache Replacement and Retention Policy

Both the L1 State and L1 Texture cache will replace a cached entry only when no invalid entries are available. A line is only replaced when no on-going sample operations are using the information of that cache-entry. If all entries are currently being used, the sampler will automatically wait until one or more lines become available. Replaced lines, since they are read-only, are invalidated.

Coherency Mechanisms

Coherency of these caches to memory must be maintained by software. There are 3 mechanisms by which the contents of the L1 State and/or L1 Texture cache are invalidated. The three possible sources for invalidation are:

- Command Streamer: Command stream can issue invalidations for state and texture independently.
- System: In cases where sampler is referencing memory which is coherent with system memory it is possible to receive an invalidation due to a snoop/invalidation from an external agent.

  This will invalidate BOTH L1 State and L1 Texture caches.
- Shader: The CACHE_FLUSH message can be generated by the shader which will invalidate BOTH the L1 State and L1 Texture caches.

From the Command Streamer it is possible to invalidate either L1 State or L1 Texture caches independently. However, invalidating only state may lead to unexpected behavior where old texture data related to an old surface or sampler state are used. It is strongly recommended that a Texture cache invalidation be done whenever a State cache invalidation is done.

See the section on 3D Sampler State for more details on how state is referenced in memory and specific programming notes.

Sampler SW Performance Recommendations

This is a section describing things that SW can do to get better performance out of sampler hardware.

I,L,LA,A formats

These are expanded to RGBA formats in sampler and as result are treated as larger texel sizes in sampler and have worse performance. Most of the time it is better to use the Shader Channel Select signals in RENDER_SURFACE_STATE to emulate them using R or RG formats.

		Channel select			
Base format	New format	R	G	B	A
I*	R*	R	R	R	R
L*	R*	R	R	R	1.0
LA*	RG*	R	R	R	G
A*	R*	0.0	0.0	0.0	R

Issues with this where result can be different. The A channel for LA and A format will be 0.0 rather than 1.0.

The table below shows the mapping requires for all Luminance, Intensity and Transparency formats:

Original Format	Mapped To Format	Channel select Programming (R/G/B/A)
A16_FLOAT	R16_FLOAT	0/0/0/R
A16_UNORM	R16_UNORM	0/0/0/R
A24X8_UNORM	R24_UNORM_X8_TYPELESS	0/0/0/R
A32_FLOAT	R32_FLOAT	0/0/0/R
A32_UNORM	R32_UNORM	0/0/0/R
A8_UNORM	R8_UNORM	0/0/0/R
I16_FLOAT	R16_FLOAT	R/R/R/R
I16_UNORM	R16_UNORM	R/R/R/R
I24X8_UNORM	R24_UNORM_X8_TYPELESS	R/R/R/R
I32_FLOAT	R32_FLOAT	R/R/R/R
I8_SINT	R8_SINT	R/R/R/R
I8_UINT	R8_UINT	R/R/R/R
I8_UNORM	R8_UNORM	R/R/R/R
L16_FLOAT	R16_FLOAT	R/R/R/1
L16_UNORM	R16_UNORM	R/R/R/1
L16A16_FLOAT	R16G16_FLOAT	R/R/R/G
L16A16_UNORM	R16G16_UNORM	R/R/R/G
L24X8_UNORM	R24_UNORM_X8_TYPELESS	R/R/R/1
L32_FLOAT	R32_FLOAT	R/R/R/1
L32_UNORM	R32_UNORM	R/R/R/1
L32A32_FLOAT	R32G32_FLOAT	R/R/R/G
L8_SINT	R8_SINT	R/R/R/1
L8_UINT	R8_UINT	R/R/R/1
L8_UNORM	R8_UNORM	R/R/R/1
L8A8_SINT	R8G8_SINT	R/R/R/G
L8A8_UINT	R8G8_UINT	R/R/R/G
L8A8_UNORM	R8G8_UNORM	R/R/R/G
Y8_UNORM	R8_UNORM	R/R/R/1

Original Format	Mapped To Format	Channel select Programming (R/G/B/A)
A16_FLOAT	R16_FLOAT	0/0/0/R
A16_UNORM	R16_UNORM	0/0/0/R
A24X8_UNORM	R24_UNORM_X8_TYPELESS	0/0/0/R
A32_FLOAT	R32_FLOAT	0/0/0/R
A32_UNORM	R32_UNORM	0/0/0/R
A8_UNORM	R8_UNORM	0/0/0/R
I16_FLOAT	R16_FLOAT	R/R/R/R
I16_UNORM	R16_UNORM	R/R/R/R
I24X8_UNORM	R24_UNORM_X8_TYPELESS	R/R/R/R
I32_FLOAT	R32_FLOAT	R/R/R/R
I8_SINT	R8_SINT	R/R/R/R
I8_UINT	R8_UINT	R/R/R/R
I8_UNORM	R8_UNORM	R/R/R/R
L16_FLOAT	R16_FLOAT	R/R/R/1
L16_UNORM	R16_UNORM	R/R/R/1
L16A16_FLOAT	R16G16_FLOAT	R/R/R/G
L16A16_UNORM	R16G16_UNORM	R/R/R/G
L24X8_UNORM	R24_UNORM_X8_TYPELESS	R/R/R/1
L32_FLOAT	R32_FLOAT	R/R/R/1
L32_UNORM	R32_UNORM	R/R/R/1
L32A32_FLOAT	R32G32_FLOAT	R/R/R/G
L8_SINT	R8_SINT	R/R/R/1
L8_UINT	R8_UINT	R/R/R/1
L8_UNORM	R8_UNORM	R/R/R/1
L8A8_SINT	R8G8_SINT	R/R/R/G
L8A8_UINT	R8G8_UINT	R/R/R/G
L8A8_UNORM	R8G8_UNORM	R/R/R/G
Y8_UNORM	R8_UNORM	R/R/R/1

Tile Mode

If in TILEMODE_LINEAR and the base_address is NOT cacheline aligned there will be a very large performance hit in addition to the one that is already on linear.

Write Channel Mask

A header is not needed to specify a channel mask other than all enabled. If head is disabled the response length is used to determine a default channel mask. See **message format** section for more details.

NativeL1 Fill rate improvement SW requirements

Due to hardware restrictions, the L1 fill rate improvements added in by these features fill is affected by Surface Horizontal Alignment. For each texel size in the L1 the following table shows the minimum value for Surface Horizontal Alignment to get full performance. Lower values do not cause a function issue but will slow down performance.

Texel size in L1	Min Horizontal Alignment
16	16
24	8
32	8
48+	4

Texture Coordinate Processing

The Texture Coordinate Processing function of the Sampling Engine performs any operations on the texture coordinates that are required before physical addresses of texel samples can be generated.

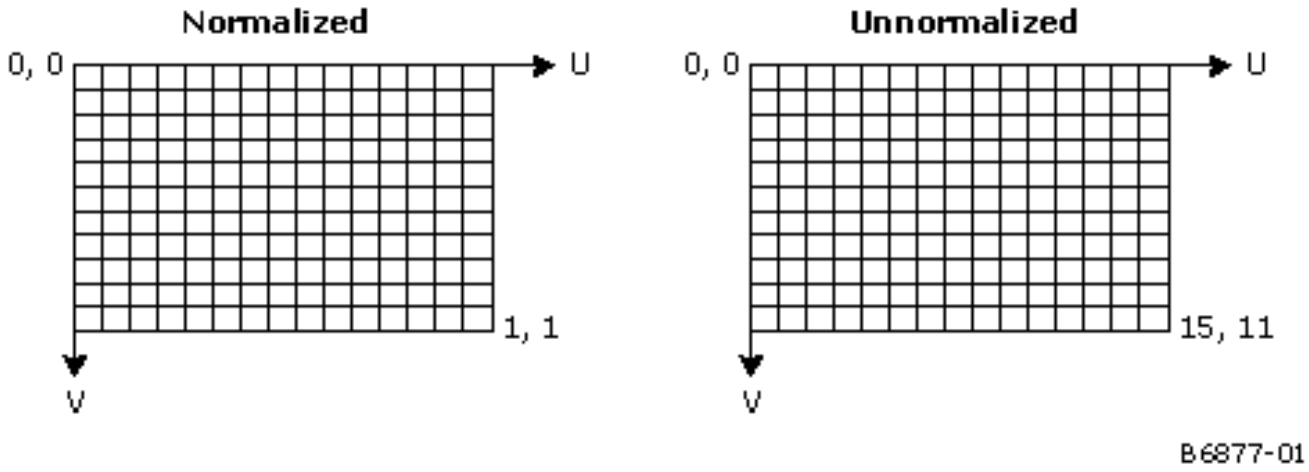
Texture Coordinate Normalization

A texture coordinate may have *normalized* or *unnormalized* values. In this function, unnormalized coordinates are normalized.

Normalized coordinates are specified in units relative to the map dimensions, where the origin is located at the upper/left edge of the upper left texel, and the value 1.0 coincides with the lower/right edge of the lower right texel . 3D rendering typically utilizes normalized coordinates.

Unnormalized coordinates are in units of texels and have not been divided (normalized) by the associated map's height or width. Here the origin is the located at the upper/left edge of the upper left texel of the base texture map.

Normalized vs. Unnormalized Texture Coordinates



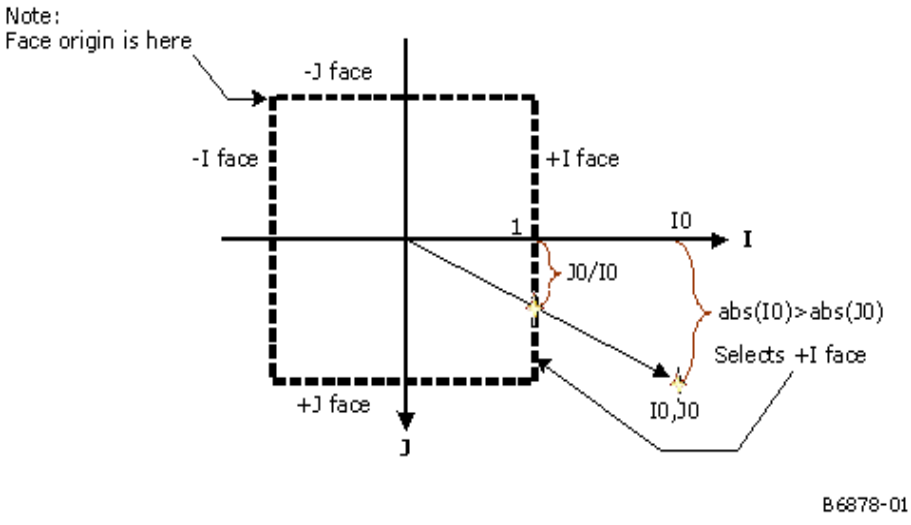
Texture Coordinate Computation

Cartesian (2D) and homogeneous (projected) texture coordinate values are projected from (interpolated) screen space back into texture coordinate space by dividing the pixel's S and T components by the Q component. This operation is done prior to sending sample operations to the 3D sampler.

Vector (cube map) texture coordinates are generated by first determining which of the 6 cube map faces (+X, +Y, +Z, -X, -Y, -Z) the vector intersects. The vector component (X, Y or Z) with the largest absolute value determines the proper (major) axis, and then the sign of that component is used to select between the two faces associated with that axis. The coordinates along the two minor axes are then divided by the coordinate of the major axis, and scaled and translated, to obtain the 2D texture coordinate ([0,1]) within the chosen face. Note that the coordinates delivered to the sampling engine must already have been divided by the component with the largest absolute value.

An illustration of this cube map coordinate computation, simplified to only two dimensions, is provided below:

Cube Map Coordinate Computation Example



Texel Address Generation

To better understand texture mapping, consider the mapping of each object (screen-space) pixel onto the textures images. In texture space, the pixel becomes some arbitrarily sized and aligned quadrilateral. Any given pixel of the object may "cover" multiple texels of the map, or only a fraction of one texel. For each pixel, the usual goal is to sample and filter the texture image in order to best represent the covered texel values, with a minimum of blurring or aliasing artifacts. Per-texture state variables are provided to allow the user to employ quality/performance/footprint tradeoffs in selecting how the particular texture is to be sampled.

The Texel Address Generation function of the Sampling Engine is responsible for determining how the texture maps are to be sampled. Outputs of this function include the number of texel to be fetched, along with the physical addresses of the samples and the filter weights to be applied to the samples after they are read. This information is computed given the incoming texture coordinate and gradient values, and the relevant state variables associated with the sampler and surface. This function also applies the texture coordinate address controls when converting the sample texture coordinates to map addresses.

Level of Detail Computation (Mipmapping)

Due to the specification and processing of texture coordinates at object vertices, and the subsequent object warping due to a perspective projection, the texture image may become *magnified* (where a texel covers more than one pixel) or *minified* (a pixel covers more than one texel) as it is mapped to an object. In the case where an object pixel is found to cover multiple texels (texture minification), merely choosing one (e.g., the texel sample nearest to the pixel's texture coordinate) will likely result in severe aliasing artifacts.

Mipmapping and texture filtering are techniques employed to minimize the effect of undersampling these textures. With mipmapping, software provides *mipmap levels,* a series of pre-filtered texture maps of decreasing resolutions that are stored in a fixed (monolithic) format in memory. When mipmaps are provided and enabled, and an object pixel is found to cover multiple texels (e.g., when a textured object is located a significant distance from the viewer), the device will sample the mipmap level(s) offering a texel/pixel ratio as close to 1.0 as possible.

The device supports up to 14 mipmap levels per map surface, ranging from 16384 x 16384 texels to a 1 X 1 texel. Each successive level has 1/2 the resolution of the previous level in the U and V directions (to a minimum of 1 texel in either direction) until a 1x1 texture map is reached. The dimensions of mipmap levels need not be a power of 2.

Each mipmap level is associated with a *Level of Detail (LOD)* number. LOD is computed as the approximate, $\log_2$ measure of the ratio of texels per pixel. The highest resolution map is considered LOD 0. A larger LOD number corresponds to lower resolution mip level.

The *Sampler[]BaseMipLevel* state variable specifies the LOD value at which the minification filter vs. the magnification filter should be applied.

When the texture map is magnified (a texel covers more than one pixel), the base map (LOD 0) texture map is accessed, and the magnification mode selects between the nearest neighbor texel or bilinear interpolation of the 4 neighboring texels on the base (LOD 0) mipmap.

Base Level Of Detail (LOD)

The per-pixel LOD is computed in an implementation-dependent manner and approximates the $\log_2$ of the texel/pixel ratio at the given pixel. The computation is typically based on the differential texel-space distances associated with a one-pixel differential distance along the screen x- and y-axes. These texel-space distances are computed by evaluating neighboring pixel texture coordinates, these coordinates being in units of texels on the base MIP level (multiplied by the corresponding surface size in texels). The q coordinates represent the third dimension for 3D (volume) surfaces, this coordinate is a constant 0 for 2D surfaces.

The ideal LOD computation is included below.

$$LOD(x,y) = \log_2[\rho(x,y)]$$

where :

$$\rho(x,y) = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial q}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial q}{\partial y}\right)^2}\right\},$$

LOD Bias

A biasing offset can be applied to the computed LOD and used to artificially select a higher or lower miplevel and/or affect the weighting of the selected mipmap levels. Selecting a slightly higher mipmap level will trade off image blurring with possibly increased performance (due to better texture cache reuse). Lowering the LOD tends to sharpen the image, though at the expense of more texture aliasing artifacts.

The LOD bias is defined as sum of the *LODBias* state variable and the *pixLODBias* input from the input message (which can be non-zero only for sample_b messages). The application of LOD Bias is unconditional, therefore these variables must both be set to zero in order to prevent any undesired biasing.

Note that, while the LOD Bias is applied prior to clamping and min/mag determination and therefore can be used to control the min-vs-mag crossover point, its use has the undesired effect of actually changing the LOD used in texture filtering.

LOD Pre-Clamping

The LOD Pre-Clamping function can be enabled or disabled via the *LODPreClampEnable* state variable.

After biasing and/or adjusting of the LOD , the computed LOD value is clamped to a range specified by the (integer and fractional bits of) *MinLOD* and *MaxLOD* state variables prior to use in Min/Mag Determination.

MaxLOD specifies the lowest resolution mip level (maximum LOD value) that can be accessed, even when lower resolution maps may be available. Note that this is the only parameter used to specify the number of valid mip levels that can be accessed, i.e., there is no explicit "number of levels stored in memory" parameter associated with a mip-mapped texture. All mip levels from the base mip level map through the level specified by the integer bits of *MaxLOD* must be stored in memory, or operation is UNDEFINED.

MinLOD specifies the highest resolution mip level (minimum LOD value) that can be accessed, where LOD==0 corresponds to the base map. This value is primarily used to deny access to high-resolution mip levels that have been evicted from memory when memory availability is low.

MinLOD and *MaxLOD* have both integer and fractional bits. The fractional parts will limit the inter-level filter weighting of the highest or lowest (respectively) resolution map. For example, if *MinLOD* is 4.5 and *MipFilter* is LINEAR, LOD 4 can contribute only up to 50% of the final texel color.

Min/Mag Determination

The biased and clamped LOD is used to determine whether the texture is being minified (scaled down) or magnified (scaled up).

The *BaseMipLevel* state variable is subtracted from the biased and clamped LOD. The *BaseMipLevel* state variable therefore has the effect of selecting the "base" mip level used to compute Min/Mag Determination. (This was added to match OpenGL semantics). Setting *BaseMipLevel* to 0 has the effect of using the highest-resolution mip level as the base map.

If the biased and clamped LOD is non-positive, the texture is being magnified, and a single (high-resolution) miplevel will be sampled and filtered using the *MagFilter* state variable. At this point the computed LOD is reset to 0.0. Note that LOD Clamping can restrict access to high-resolution miplevels.

If the biased LOD is positive, the texture is being minified. In this case the *MipFilter* state variable specifies whether one or two mip levels are to be included in the texture filtering, and how that (or those) levels are to be determined as a function of the computed LOD.

LOD Computation Pseudocode

This section illustrates the LOD biasing and clamping computation in pseudocode, encompassing the steps described in the previous sections. The computation of the initial per-pixel LOD value *LOD* is not shown.

```
Bias:        S4.8
 MinLod:      U4.8
 MaxLod:      U4.8
 Base:        U4.1
 MIPCnt:      U4
 SurfMinLod: U4
 ResMinLod:   U4.8

 PerSampleMinLOD: float32

 MinLod         = max(MinLod, PerSampleMinLOD)
 AdjMaxLod      = min(MaxLod, MIPCnt)
 AdjMinLod      = min(MinLod, MIPCnt)
 AdjPR_minLOD   = ResMinLod - SurfMinLod
 AdjMinLod      = max(AdjMinLod, AdjPR_minLOD)
 Out_of_Bounds = AdjPR_minLOD > MIPCnt

 if ( sample_b )
     LOD += Bias + bias_parameter
 else if ( sample_l or ld )
     LOD = Bias + lod_parameter
 else
```

```
     LOD += Bias

     PreClamp = LODPreClampMode != PRECLAMP_NONE
     if ( PreClamp )
         if ( PRECLAMP_D3D )
             LOD = min(LOD, AdjMaxLod)
             LOD = max(LOD, AdjMinLod)
         else
             LOD = min(LOD, MaxLod)
             LOD = max(LOD, MinLod)

 MagMode = (LOD - Base <= 0)

MagClampMipNone = LODClampMagnificationMode == MAG_CLAMP_MIPNONE

 if ( (MagMode && MagClampMipNone) or MipFlt == None )
     LOD = 0
     LOD = min(LOD, ceil(AdjMaxLod))
     LOD = max(LOD, floor(AdjMinLod))
 else if ( MipFlt == Nearest )

LOD = min(LOD, ceil(AdjMaxLod))
 LOD = max(LOD, floor(AdjMinLod))

     LOD += 0.5
     LOD = floor(LOD)
 else
     // MipFlt = Linear
     LOD = min(LOD, AdjMaxLod)
     LOD = max(LOD, AdjMinLod)
     TriBeta = frac(LOD)
     LOD0 = floor(LOD)
     LOD1 = LOD0 + 1

 if ( ! lod )  // "LOD" message type
     Lod += SurfMinLod
```

If Out_of_Bounds is true, LOD is set to zero and instead of sampling the surface the texels are replaced with zero in all channels, except for surface formats that don't contain alpha, for which the alpha channel is replaced with one. These texels then proceed through the rest of the pipeline.

Intra-Level Filtering Setup

Depending on whether the texture is being minified or magnified, the *MinFilter* or *MagFilter* state variable (respectively) is used to select the sampling filter to be used within a mip level (intra-level, as opposed to any inter-level filter). Note that for volume maps, this selection also applies to filtering between layers.

The processing at this stage is restricted to the selection of the filter type, computation of the number and texture map coordinates of the texture samples, and the computation of any required filter parameters. The filtering of the samples occurs later on in the Sampling Engine function.

The following table summarizes the intra-level filtering modes.

Sampler Min/ MagFilter value	Description
MAPFILTER_NEAREST	Supported on all surface types. The texel nearest to the pixel's U,V,Q coordinate is read and output from the filter.
MAPFILTER_LINEAR	Not supported on buffer surfaces. The 2, 4, or 8 texels (depending on 1D, 2D/CUBE, or 3D surface, respectively) surrounding the pixel's U,V,Q coordinate are read and a linear filter is applied to produce a single filtered texel value.
MAPFILTER_ANISOTROPIC	Not supported on buffer or 3D surfaces. A projection of the pixel onto the texture map is generated and "subpixel" samples are taken along the major axis of the projection (center axis of the longer dimension). The outermost subpixels are weighted according to closeness to the edge of the projection, inner subpixels are weighted equally. Each subpixel samples a bilinear 2x2 of texels and the results are blended according to weights to produce a filtered texel value.
MAPFILTER_MONO	Supported only on 2D surfaces. This filter is only supported with the monochrome (MONO8) surface format. The monochrome texel block of the specified size surrounding the pixel is selected and filtered.

MAPFILTER_NEAREST

When the MAPFILTER_NEAREST is selected, the texel with coordinates nearest to the pixel's texture coordinate is selected and output as the single texel sample coordinates for the level. This is a form of "Point Sampling".

Corner Texel Mode

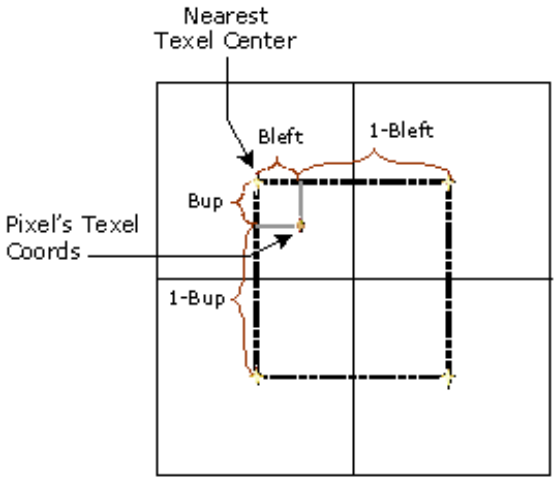
When **Corner Texel Mode** is enabled in the RENDER_SURFACE_STATE the definition of how the texel location is calculated is modified. Corner Texel Mode shifts the edge of the map to be coincident with the center of the edge texels. The 3D Sampler will handle this shift when sampling the texel.

MAPFILTER_LINEAR

The following description indicates behavior of the MIPFILTER_LINEAR filter for 2D and CUBE surfaces. 1D and 3D surfaces follow a similar method but with a different number of dimensions available.

When the MAPFILTER_LINEAR filter is selected on a 2D surface, the 2x2 region of texels surrounding the pixel's texture coordinate are sampled and later bilinearly filtered. The filter weights each texel's contribution according to its distance from the pixel center. Texels further from the pixel center receive a smaller weight.

Bilinear Filter Sampling

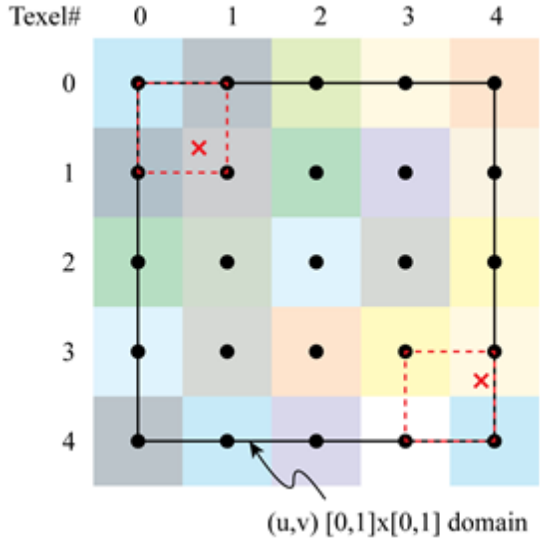


B6879-01

Corner Texel Mode

The Corner Texel Mode bit in surface state changes the linear filtering, by shifting the edges of the map to the center of the texel. Effectively, this means

that the edge of the map no coincides with the center of the texel, hence the name "Corner Texel Mode". The diagram below shows this.



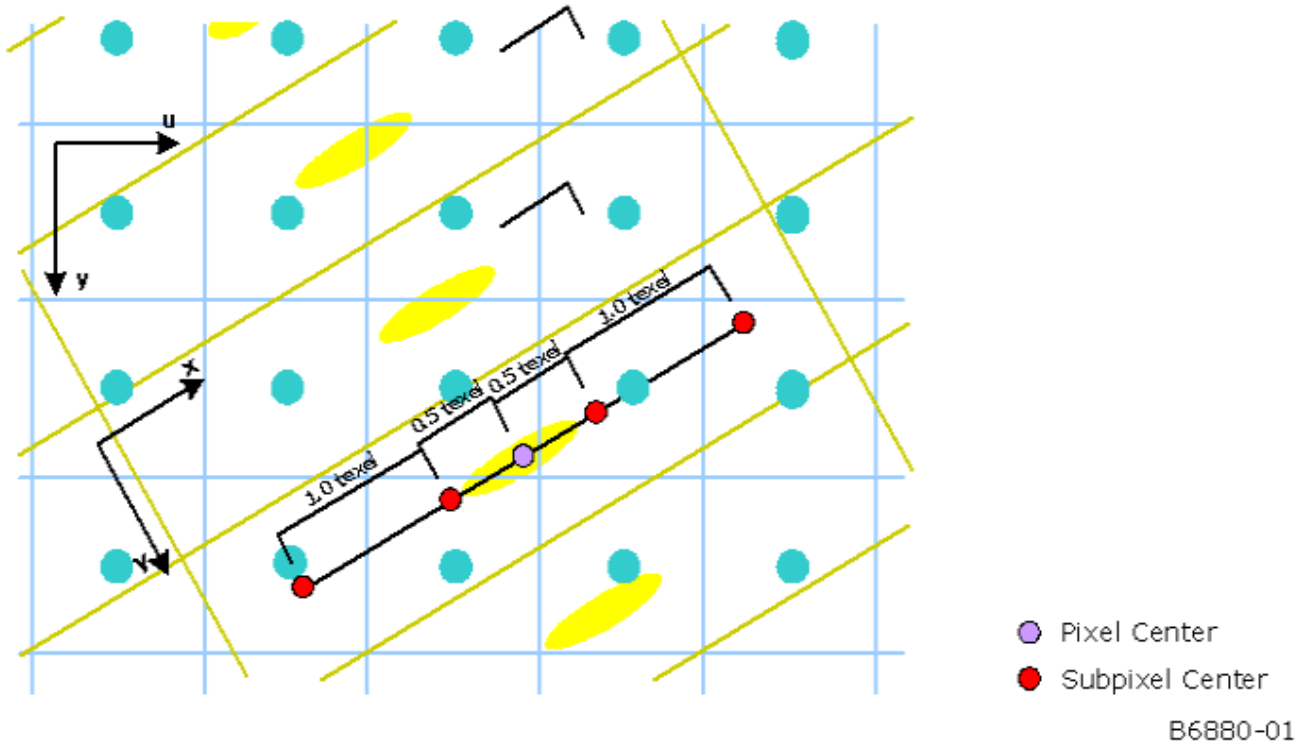
When doing linear filtering, the 3D sampler will treat a surface as being 1 texel less in each dimension when calculating texel location from the sample coordinates.

The benefit of this mode is that it is guaranteed that all the texels which contribute to the linear filter of a sample will be on the map if the sample point is on the map.

MAPFILTER_ANISOTROPIC

The MAPFILTER_ANISOTROPIC texture filter attempts to compensate for the anisotropic mapping of pixels into texture map space. A possibly non-square set of texel sample locations will be sampled and later filtered. The *MaxAnisotropy* state variable is used to select the maximum aspect ratio of the filter employed, up to 16:1.

The algorithm employed first computes the major and minor axes of the pixel projection onto the texture map. LOD is chosen based on the minor axis length in texel space. The anisotropic "ratio" is equal to the ratio between the major axis length and the minor axis length. The next larger even integer above the ratio determines the anisotropic number of "ways", which determines how many subpixels are chosen. A line along the major axis is determined, and "subpixels" are chosen along this line, spaced one texel apart, as shown in the diagram below. In this diagram, the texels are shown in light blue, and the pixels are in yellow.



Each subpixel samples a bilinear 2x2 around it just as if it was a single pixel. The result of each subpixel is then blended together using equal weights on all interior subpixels (not including the two endpoint subpixels). The endpoint subpixels have lesser weight, the value of which depends on how close the "ratio" is to the number of "ways". This is done to ensure continuous behavior in animation.

Inter-Level Filtering Setup

The *MipFilter* state variable determines if and how texture mip maps are to be used and combined. The following table describes the various mip filter modes:

MipFilter Value	Description
MIPFILTER_NONE	Mipmapping is DISABLED. Apply a single filter on the highest resolution map available (after LOD clamping).
MIPFILTER_NEAREST	Choose the nearest mipmap level and apply a single filter to it. Here the biased LOD will be rounded to the nearest integer to obtain the desired miplevel. LOD Clamping may further restrict this miplevel selection.
MIPFILTER_LINEAR	Apply a filter on the two closest mip levels and linear blend the results using the distance between the computed LOD and the level LODs as the blend factor. Again, LOD Clamping may further restrict the selection of miplevels (and the blend factor between them).

When minifying and MIPFILTER_NEAREST is selected, the computed LOD is rounded to the nearest mip level.

When minifying and MIPFILTER_LINEAR is selected, the fractional bits of the computed LOD are used to generate an inter-level blend factor. The LOD is then truncated. The mip level selected by the truncated LOD, and the next higher (lower resolution) mip level are determined.

Regardless of *MipFilter* and the min/mag determination, all computed LOD values (two for MIPFILTER_LINEAR, otherwise one) are then unconditionally clamped to the range specified by the (integer bits of) *MinLOD* and *MaxLOD* state variables.

Texture Address Control

The *[TCX,TCY,TCZ]ControlMode* state variables control the access and/or generation of texel data when the specific texture coordinate component falls *outside* of the normalized texture map coordinate range [0,1).

The table below provides all the supported Address Control modes for each direction.

TC[X,Y,Z] Control	Operation
TEXCOORDMODE_CLAMP	Clamp to the texel value at the edge of the map.
TEXCOORDMODE_CLAMP_BORDER	Use the texture map's border color for any texel samples falling outside the map. The border color is specified via a pointer in SAMPLER_STATE.
TEXCOORDMODE_HALF_BORDER	Similar to CLAMP_BORDER except texels outside of the map are clamped to a value halfway between the edge texel and the border color.
TEXCOORDMODE_WRAP	Upon crossing an edge of the map, repeat at the other side of the map in the same dimension.
TEXCOORDMODE_CUBE	Only used for cube maps. Here texels from adjacent cube faces can be sampled along the edges of faces. This is considered the highest quality mode for cube environment maps.
TEXCOORDMODE_MIRROR	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture

TC[X,Y,Z] Control	Operation
	coordinates.
TEXCOORDMODE_MIRROR_ONCE	Similar to the wrap mode, though reverse direction through the map each time an edge is crossed. INVALID for use with unnormalized texture coordinates.
TEXCOORDMODE_MIRROR_101	Similar to MIRROR_ONCE in that it only reflects once in each direction, the difference is that it will skip the first pixel of the reflected image.

Separate controls are provided for texture TCX, TCY, TCZ coordinate components so, for example, the TCX coordinate can be wrapped while the TCY coordinate is clamped. Note that there are no controls provided for the TCW component as it is only used to scale the other 3 components before addressing modes are applied.

Programming Note	
Context:	Texture Address Control
TEXCOORDMODE_CUBE can only be used with SURFTYPE_CUBE	

Maximum Wraps/Mirrors

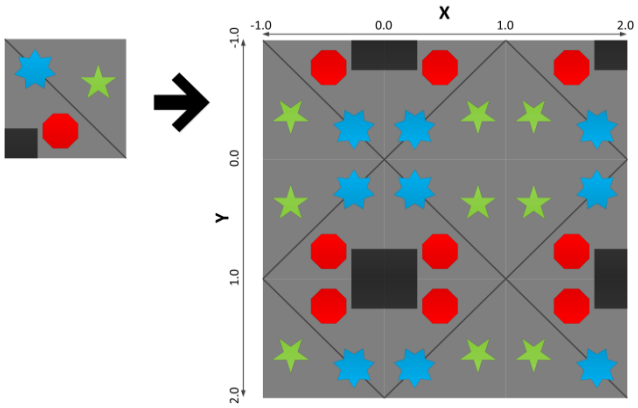
The number of map wraps on a given object is limited to 32. Going beyond this limit is legal, but may result in artifacts due to insufficient internal precision, especially evident with larger surfaces. Precision loss starts at the subtexel level (slight color inaccuracies) and eventually reaches the texel level (choosing the wrong texels for filtering).

Note: For **Wrap Shortest** mode, the setup kernel has already taken care of correctly interpolating the texture coordinates. Software needs to specify TEXCOORDMODE_WRAP mode for the sampler that is provided with wrap-shortest texture coordinates, or artifacts may be generated along map edges.

TEXCOORDMODE_MIRROR Mode

TEXCOORDMODE_MIRROR addressing mode is similar to Wrap mode, though here the base map is flipped at every integer junction. For example, for U values between 0 and 1, the texture is addressed normally, between 1 and 2 the texture is flipped (mirrored), between 2 and 3 the texture is normal again, and so on. The second row of pictures in the figure below indicate a map that is mirrored in one direction and then both directions. You can see that in the mirror mode every other integer map wrap the base map is mirrored in either direction.

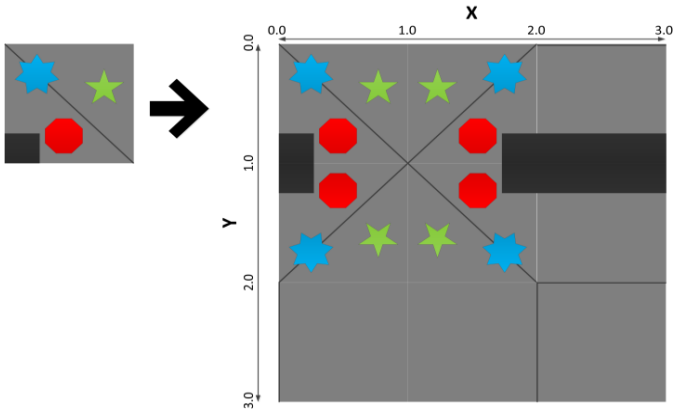
The example below shows how a simple 2D map with TEXCOORDMODE_MIRROR for both TCX and TCY is mapped.



TEXCOORDMODE_MIRROR_ONCE Mode

The TEXCOORDMODE_MIRROR_ONCE addressing mode is a combination of Mirror and Clamp modes. The absolute value of the texture coordinate component is first taken (thus mirroring about 0), and then the result is clamped to 1.0. The map is therefore mirrored once about the origin, and then clamped thereafter. This mode is used to reduce the storage required for symmetric maps.

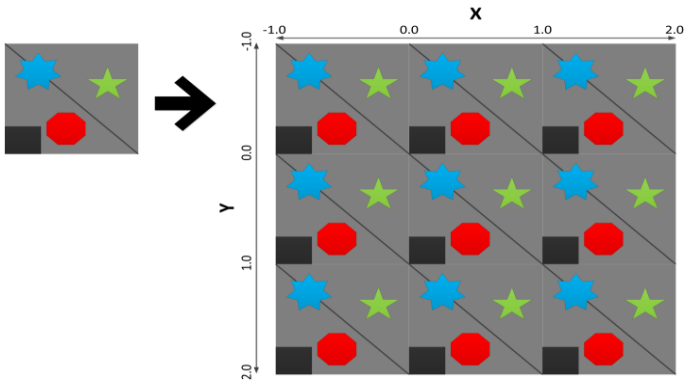
The example below shows how a simple 2D map with TEXCOORDMODE_MIRROR_ONCE for both TCX and TCY is mapped.



TEXCOORDMODE_WRAP Mode

In TEXCOORDMODE_WRAP addressing mode, the integer part of the texture coordinate is discarded, leaving only a fractional coordinate value. This results in the effect of the base map ([0,1)) being continuously repeated in all (axes-aligned) directions. Note that the interpolation between coordinate values 0.1 and 0.9 passes through 0.5 (as opposed to WrapShortest mode which interpolates through 0.0).

The example below shows how a simple 2D map with TEXCOORDMODE_WRAP for both TCX and TCY is mapped.

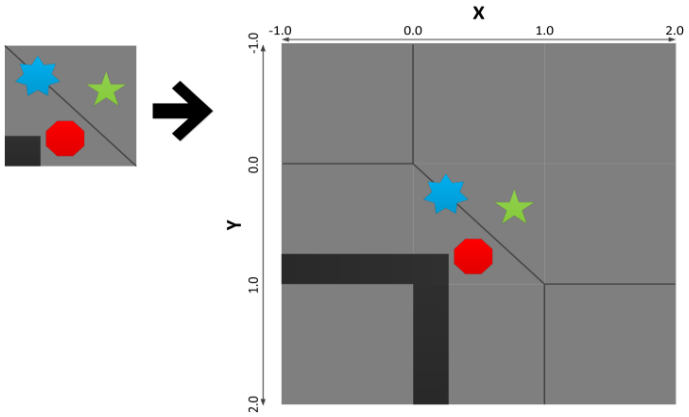


TEXCOORDMODE_CLAMP Mode

The TEXCOORDMODE_CLAMP addressing mode repeats the "edge" texel when the texture coordinate extends outside the [0,1) range of the base texture map. This is contrasted to TEXCOORDMODE_CLAMPBORDER mode which defines a separate texel value for off-map samples. TEXCOORDMODE_CLAMP is also supported for cube maps, where texture samples will only be obtained from the intersecting face (even along edges).

The figure below illustrates the effect of clamp mode. The base texture map is shown, along with a texture mapped object with texture coordinates extending outside of the base map region.

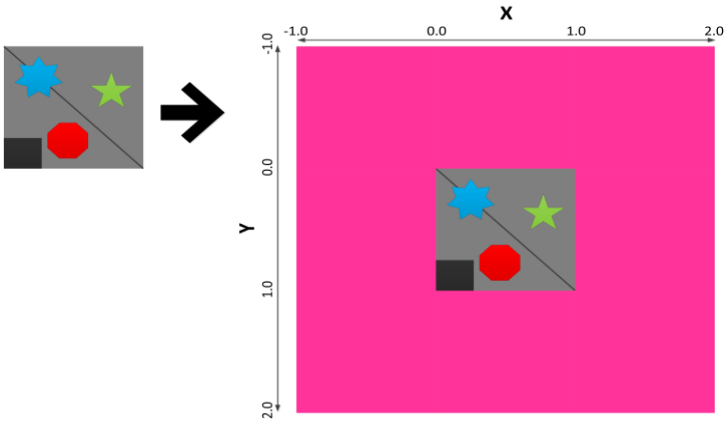
Texture Clamp Mode



TEXCOORDMODE_CLAMPBORDER Mode

For non-cube map textures, TEXCOORDMODE_CLAMPBORDER addressing mode specifies that the texture map's border value *BorderColor* is to be used for any texel samples that fall outside of the base map. The border color is specified via a pointer in SAMPLER_STATE.

The example below shows how a simple 2D map with TEXCOORDMODE_CLAMPBORDER for both TCX and TCY is mapped.



TEXCOORDMODE_HALF_BORDER Mode

For non-cube map textures, TEXCOORDMODE_HALF_BORDER addressing mode is similar to the TEXCOORDMODE_CLAMPBORDER in that it specifies that the texture map's border value *BorderColor*. However this value is blended with the edge texel color and used for any texel samples that fall outside of the base map. The border color is specified via a pointer in SAMPLER_STATE.

TEXCOORDMODE_CUBE Mode

For cube map textures TEXCOORDMODE_CUBE addressing mode can be set to allow inter-face filtering. When texel sample coordinates that extend beyond the selected cube face (e.g., due to intra-level filtering near a cube edge), the correct sample coordinates on the adjoining face will be computed. This will eliminate artifacts along the cube edges, though some artifacts at cube corners may still be present.

TEXCOORDMODE_MIRROR_101

The MIRROR_101 address control mode is similar to the MIRROR_ONCE. The difference is that it will skip the first pixel of the reflected image. Effectively, this means that the reflected images (in both directions) are one pixel smaller in size.

Programming Note	
Context:	MIRROR_101 and Quilted Surfaces
MIRROR_101 is not supported for Quilted Surfaces.	

Programming Note	
Context:	Fast Anisotropic Filtering
MIROR_101 Texture Coordinate Mode cannot be used with Anisotropic Filtering.	

Texel Fetch

The Texel Fetch function of the Sampling Engine reads the texture map contents specified by the texture addresses associated with each texel sample. The texture data is read either directly from the memory-resident texture map, or from internal texture caches. The texture caches can be invalidated by the

Sampler Cache Invalidate field of the MI_FLUSH instruction or via the **Read Cache Flush Enable** bit of PIPE_CONTROL. Except for consideration of coherency with CPU writes to textures and rendered textures, the texture cache does not affect the functional operation of the Sampling Engine pipeline.

When the surface format of a texture is defined as being a compressed surface, the Sampler will automatically decompress from the stored format into the appropriate [A]RGB values. The compressed texture storage formats and decompression algorithms can be found in the *Memory Data Formats* chapter. When the surface format of a texture is defined as being an index into the texture palette (format names including "Px"), the palette lookup of the index determines the appropriate RGB values.

Texel Chroma Keying

ChromaKey is a term used to describe a method of effectively removing or replacing a specific range of texel values from a map that is applied to a primitive, e.g., in order to define transparent regions in an RGB map. The Texel Chroma Keying function of the Sampling Engine pipeline conditionally tests texel samples against a "key" range, and takes certain actions if any texel samples are found to match the key.

Chroma Key Testing

ChromaKey refers to testing the texel sample components to see if they fall within a range of texel values, as defined by *ChromaKey[][High,Low]* state variables. If each component of a texel sample is found to lie within the respective (inclusive) range and ChromaKey is enabled, then an action will be taken to remove this contribution to the resulting texel stream output. Comparison is done separately on each of the channels and only if all 4 channels are within range the texel will be eliminated.

The Chroma Keying function is enabled on a per-sampler basis by the *ChromaKeyEnable* state variable.

The *ChromaKey[][High,Low]* state variables define the tested color range for a particular texture map.

Chroma Key Effects

There are two operations that can be performed to "remove" matching texel samples from the image. The *ChromaKeyEnable* state variable must first enable the chroma key function. The *ChromaKeyMode* state variable then specifies which operation to perform on a per-sampler basis.

The *ChromaKeyMode* state variable has the following two possible values:

KEYFILTER_KILL_ON_ANY_MATCH: Kill the pixel if any contributing texel sample matches the key.

KEYFILTER_REPLACE_BLACK: Here the sample is replaced with (0,0,0,0).

The Kill Pixel operation has an effect on a pixel only if the associated sampler is referenced by a sample instruction in the pixel shader program. If the sampler is not referenced, the chroma key compare is not done and pixels cannot be killed based on it.

Shadow Prefilter Compare

When a *sample_c* message type is processed, a special shadow-mapping precomparison is performed on the texture sample values prior to filtering. Specifically, each texture sample value is compared to the "ref" component of the input message, using a compare function selected by *ShadowFunction*, and

described in the table below. Note that only single-channel texel formats are supported for shadow mapping, and so there is no specific color channel on which the comparison occurs.

ShadowFunction	Result
PREFILTEROP_ALWAYS	0.0
PREFILTEROP_NEVER	1.0
PREFILTEROP_LESS	(texel < ref) ? 0.0 : 1.0
PREFILTEROP_EQUAL	(texel == ref) ? 0.0 : 1.0
PREFILTEROP_LEQUAL	(texel <= ref) ? 0.0 : 1.0
PREFILTEROP_GREATER	(texel > ref) ? 0.0 : 1.0
PREFILTEROP_NOTEQUAL	(texel != ref) ? 0.0 : 1.0
PREFILTEROP_GEQUAL	(texel >= ref) ? 0.0 : 1.0

The binary result of each comparison is fed into the subsequent texture filter operation (in place of the texel's value which would normally be used).

Software is responsible for programming the "ref" component of the input message such that it approximates the same distance metric programmed in the texture map (e.g., distance from a specific light to the object pixel). In this way, the comparison function can be used to generate "in shadow" status for each texture sample, and the filtering operation can be used to provide soft shadow edges.

Texel Filtering

The Texel Filtering function of the Sampling Engine performs any required filtering of multiple texel values on and possibly between texture map layers and levels. The output of this function is a single texel color value.

The state variables *MinFilter*, *MagFilter*, and *MipFilter* are used to control the filtering of texel values. The *MipFilter* state variable specifies how many mipmap levels are included in the filter, and how the results of any filtering on these separate levels are combined to produce a final texel color. The *MinFilter* and *MagFilter* state variables specify how texel samples are filtered within a level.

Texel Color Gamma Linearization

This function is supported to allow pre-gamma-corrected texel RGB (not A) colors to be mapped back into linear (gamma=1.0) gamma space prior to (possible) blending with and writing to the Color Buffer. This permits higher quality image blending by performing the blending on colors in linear gamma space.

This function is enabled on a per-texture basis by use of a surface format with "_SRGB" in its name. If enabled, the pre-filtered texel RGB color to be converted to gamma=1.0 space by applying a ^(2.4) exponential function.

Multisampled Surface Behavior

Multisampled surfaces are sampled using a dedicated point-sample message **ld2dms** or **l2dms_w**. These messages contain Si (sample index) which is used to sample from a specific subsample plane. Each texel of a multi-sampled surface has 1,2,4,8 or 16 "subsamples" per texel.

The sampleinfo message returns specific parameters associated with a multisample surface such as the number of subsamples per texel. The resinfo message returns the height, width, depth (in units of *pixels*, not samples), and MIP count of the surface.

Multisampled surfaces typically have an associated MCS control surface which contains information about which indicates which sample index contains the color data for a specific subample. The control surface can be accessed using the **ld_mcs** message, which returns the information for all subsamples for a given texel.

From an optimization perspective there are two different approaches which can be used to resolver a multisampled surface:

A shader can fetch ld_mcs to determine which subsamples need to be fetched using ld2dms or ld2dms_w. In general, it is expected that most subsamples will reference Sample index zero for it's color by default. It may be optimal for the shader to always fetch to sample index 0 unconditionally, and only fetch to other sample indices based on the fetched MCS. Resolve can then be done with the minimal number of point-sample operations.

Alternatively, a shader can unconditionally fetch all subsamples to minimize dependencies. In this case the MCS data returned from the ld_mcs is simply passed to the ld2dms message.

Multisample Control Surface

Four messages have been defined for the sampling engine, *ld_mcs*, *ld2dms*, *ld2dms_w* and *ld2dss*. A pixel shader kernel sampling from an multisampled surface using an MCS must first sample from the MCS surface using the *ld_mcs* message. This message behaves like the *ld* message, except that the surface is defined by the MCS fields of SURFACE_STATE rather than the normal fields. The surface format is effectively R8_UINT for 4x surfaces, R32_UINT for 8x surfaces, and two R32_UNIT for x16 surfaces thus data is returned in unsigned integer format. Following the *ld_mcs*, the kernel issues a *ld2dms* or ld2dms_w message to sample the surface itself. The integer value from the MCS surface is delivered in the mcs parameter of this messages as the sample index.

Since *sample* is no longer supported on multisampled surfaces, the multisample resolve must be done using *ld2dms or ld2dms_w*. For surfaces with **Multisampled Surface Storage Format** set to MSFMT_MSS and **MCS Enable** set to enabled, an optimization is available to enable higher performance for compressed pixels. The *ld2dss* message can be used to sample from a particular sample slice on the surface. By examining the MCS value, software can determine which sample slices to sample from. A simple optimization with probable large return in performance is to compare the MCS value to zero (indicating all samples are on sample slice 0), and sample only from sample slice 0 using *ld2dss* if MCS is zero. Sample slice 0 is the pixel color in this case. If MCS is not zero, each sample is then obtained using *ld2dms* messages and the results are averaged in the kernel after being returned. Refer to the multisample storage format in the GPU Overview volume for more details.

State

SW Generated BINDING_TABLE_STATE

The 3D sampler uses both surface state objects (RENDER_SURFACE_STATE) as well as sampler state objects (SAMPLER_STATE). These objects are cached locally in the sampler state cache

for improved performance as it is assumed that many sampler messages will utilize the same surface and sampler states.

Programming Note	
Context:	Out of Bounds Handling
If a pointer to sampler or surface state goes beyond the end of the sampler or surface state buffer (as defined by the associated size field of the STATE_BASE_ADDRESS command) the sampler will force the address offset to cache-line 0 from the defined Base Address. The result of this state fetch is undefined and depends on how the state buffer has been populated.	

Surface State Fetch

Surface state is fetched from system memory using a Binding Table Pointer (**BTP**). The **BTP** is a 16-bit value provided by the command stream (not directly by the shader) which determines the binding-table to be used. An 8-bit Binding Table Index (**BTI**) is then provided by the shader via the message descriptor, which indicates the offset into the Binding Table. The BTP and BTI are relative to the **Surface State Base Address** and the binding table itself resides in system memory. The contents of the Binding Table is a list of pointers to surface state objects. The pointer from the Binding Table is also relative to the **Sampler State Base Address,** and points directly to a 256-bit **RENDER_SURFACE_STATE** object which sampler will fetch and store in its internal state cache.

Bindless Surface State Fetch

The sampler supports a "Bindless" surface model. Bindless refers to the fact that a Binding table is not required for this mode. Instead, the shader sets the BTI to 253 in the descriptor to indicate Bindless and provides a 20-bit **Surface State Offset (SSO)** from the **Bindless Surface State Base Address.** This SSO directly selects a specific surface state object which is cached by the 3D Sampler in local State Cache. Both bindless and non-bindless (legacy) modes can be operated at the same time by the sampler.

Sampler State Fetch

SAMPLER_STATE objects are fetched independently of surface state and cached locally in the 3D sampler independently (there may one or more SAMPLER_STATE objects associate with one or more RENDER_SURFACE_STATE objects). The sampler state is fetched using the **Sampler State Pointer (SSP)** which is provided either in the message header or directly from the command stream (message headers are not required). The **SSP** is an offset relative to the **Dynamic_State_Base_Address** and selects a table of 16 sampler states. The 4-bit **Sampler Index** (**SI**) in the message descriptor is used to select the specific **SAMPLER_STATE** object to be fetched from system memory and cached locally in the 3D sampler.

Bindless Sampler State

The sampler supports a "Bindless" sampler model. Bindless in this case does not actually refer to the lack of a Binding table since the legacy Sampler State model also did not have a Binding Table. However, the mechanism is similar to bindless surfaces in that the pointer provided directly selects a SAMPLER_STATE object. The sampler uses the same **Sampler State Pointer (SSP)**, but it is relative to the

Bindless_Sampler_Base_Address. The Sampler Index (SI) in the message descriptor is not used, and can be set to 0. The SAMPLER_STATE object is cached locally in the 3D sampler.

State Caching

As mentioned above, the 3D Sampler allows for automatic caching of **RENDER_SURFACE_STATE** objects and **SAMPLER_STATE** objects to provide higher performance. Coherency with system memory in the state cache, like the texture cache is handled partially by software. It is expected that the command stream or shader will issue Cache Flush operation or Cache_Flush sampler message to ensure that the L1 cache remains coherent with system memory.

Programming Note	
Context:	State Cache Coherency
Whenever the value of the Dynamic_State_Base_Addr, Surface_State_Base_Addr, or Binding_Table_Pool_Base_Addr (if Binding Table Pool is enabled) are altered, the L1 state cache and L1 texture cache must be invalidated to ensure the new surface or sampler state is fetched from system memory. Whenever the RENDER_SURFACE_STATE object in memory pointed to by the **Binding Table Pointer (BTP)** and **Binding Table Index (BTI)** is modified or SAMPLER_STATE object pointed to by the **Sampler State Pointer (SSP)** and **Sampler Index (SI)** is modified, the L1 state cache and L1 texture cache must be invalidated to ensure the new surface or sampler state is fetched from system memory.	

Programming Note	
Context:	Bindless Surface State Cache Coherency
Whenever the value of the Bindless_Surface_State_Base_Addr is altered, the L1 state cache, and L1 texture cache must be invalidated to ensure the new surface or sampler state is fetched from system memory. Whenever the RENDER_SURFACE_STATE object in memory pointed to by the **Surface State Offset (SSO)** is modified, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory.	

Programming Note	
Context:	Bindless Sampler State Cache Coherency
Whenever the value of the Bindless_Sampler_State_Base_Addr is altered, the L1 state cache and L1 texture cache must be invalidated to ensure the new surface or sampler state is fetched from system memory. Whenever the SAMPLER_STATE object pointed to by the **Sampler State Pointer (SSP)** is modified, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory.	

Programming Note	
Context:	State Coherency for indirect values
Any values referenced by pointers within the RENDER_SURFACE_STATE or SAMPLER_STATE (e.g., Clear Color Pointer, Border Color or Indirect State Pointer) are considered to be part of that state and any changes to these referenced values requires an invalidation of the L1 state cache to ensure the new values are being used as part of the state. In the case of surface data pointed to by the Surface Base Address in RENDER SURFACE STATE, the Texture Cache must be invalidated if the surface data changes.	

SURFACE_STATE

The surface state is stored as individual elements, each with its own pointer in the binding table or its own entry in a memory heap in memory. Each surface state element is aligned to a 32-byte boundary.

Surface state defines the state needed for the following objects:

- texture maps (1D, 2D, 3D, cube) read by the sampling engine
- buffers read by the sampling engine
- constant buffers read by the data cache via the data port
- render targets read/written by the render cache via the data port
- streamed vertex buffer output written by the render cache via the data port
- media surfaces read from the texture cache or render cache via the data port
- media surfaces written to the render cache via the data port

The Render Surface State definition can be found in the following section:

RENDER_SURFACE_STATE

Surface Formats

The RENDER_SURFACE_STATE contains a 9-bit field called **Surface Format**, which defines the exact format of the surface being sampled. The definition of the encodings for each supported format, including compressed formats, can be found in the following section:

SURFACE_FORMAT

Programming Note	
Context:	Luminance, Intensity and Transparency Formats
Luminance, Intensity and Opacity formats can be more efficiently supported with higher performance by mapping them to their equivalent RGB format and programming the Shader Channel selects in the RENDER_SURFACE_STATE. See **SW Performance Hints** for a detailed description of this usage model.	

Sampler-Support Surface Formats

The table below lists all the surface formats which are supported by the Sampler for filtering and any restrictions.

The following table also indicates the mapping of the channels from the surface to the channels output from the sampling engine.

Some formats are supported only in DX10/OGL **Border Color Mode**. Those formats have only that mode indicated. Formats that behave the same way in both **Border Color Modes** are indicated by that column being blank.

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
R32G32B32A32_FLOAT				R	G	B	A					
R32G32B32A32_SINT			DX10/OGL	R	G	B	A					
R32G32B32A32_UINT			DX10/OGL	R	G	B	A					
R32G32B32X32_FLOAT				R	G	B	1.0					
R32G32B32_FLOAT				R	G	B	1.0					
R32G32B32_SINT			DX10/OGL	R	G	B	1.0					
R32G32B32_UINT			DX10/OGL	R	G	B	1.0					
R16G16B16A16_UNORM		Yes		R	G	B	A					
R16G16B16A16_SNORM				R	G	B	A					
R16G16B16A16_SINT			DX10/OGL	R	G	B	A					
R16G16B16A16_UINT			DX10/OGL	R	G	B	A					
R16G16B16A16_FLOAT				R	G	B	A					
R32G32_FLOAT			DX10/OGL	R	G	0.0	1.0	OCL	R	G	1.0	1.0
R32G32_SINT			DX10/OGL	R	G	0.0	1.0	OCL				
R32G32_UINT			DX10/OGL	R	G	0.0	1.0	OCL				
R32_FLOAT_X8X24_TYPELESS	Yes		DX10/OGL	R	0.0	0.0	1.0	OCL				
X32_TYPELESS_G8X24_UINT			DX10/OGL	0.0	G	0.0	1.0	OCL				
L32A32_FLOAT			DX10/OGL	L	L	L	A					
R16G16B16X16_UNORM				R	G	B	1.0					
R16G16B16X16_FLOAT				R	G	B	1.0					
A32X32_FLOAT				0.0	0.0	0.0	A					
L32X32_FLOAT				L	L	L	1.0					
I32X32_FLOAT				I	I	I	I					
B8G8R8A8_UNORM		Yes		R	G	B	A					
B8G8R8A8_UNORM_SRGB				R	G	B	A					
R10G10B10A2_UNORM				R	G	B	A					
R10G10B10A2_UINT			DX10/OGL	R	G	B	A					
R10G10B10_SNORM_A2_UNORM				R	G	B	A					
R10G10B10_FLOAT_A2_UNORM			DX10/OGL	R	G	B	A					
R8G8B8A8_UNORM				R	G	B	A					

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
R8G8B8A8_UNORM_SRGB				R	G	B	A					
R8G8B8A8_SNORM				R	G	B	A					
R8G8B8A8_SINT			DX10/OGL	R	G	B	A					
R8G8B8A8_UINT			DX10/OGL	R	G	B	A					
R16G16_UNORM		Yes	DX10/OGL	R	G	0.0	1.0	OCL	R	G	1.0	1.0
R16G16_SNORM			DX10/OGL	R	G	0.0	1.0	OCL	R	G	1.0	1.0
R16G16_SINT			DX10/OGL	R	G	0.0	1.0	OCL				
R16G16_UINT			DX10/OGL	R	G	0.0	1.0	OCL				
R16G16_FLOAT			DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
B10G10R10A2_UNORM				R	G	B	A					
R11G11B10_FLOAT				R	G	B	1.0					
R32_SINT			DX10/OGL	R	0.0	0.0	1.0	OCL				
R32_UINT			DX10/OGL	R	0.0	0.0	1.0	OCL				
R32_FLOAT	Yes		DX10/OGL	R	0.0	0.0	1.0	OCL	R	1.0	1.0	1.0
R24_UNORM_X8_TYPELESS	Yes		DX10/OGL	R	0.0	0.0	1.0	OCL				
X24_TYPELESS_G8_UINT			DX10/OGL	0.0	G	0.0	1.0					
L16A16_UNORM				L	L	L	A					
I24X8_UNORM				I	I	I	I					
L24X8_UNORM				L	L	L	1.0					
A24X8_UNORM				0.0	0.0	0.0	a					
I32_FLOAT				I	I	I	I					
L32_FLOAT				L	L	L	1.0					
A32_FLOAT				0.0	0.0	0.0	A					
B8G8R8X8_UNORM		Yes		R	G	B	1.0					
B8G8R8X8_UNORM_SRGB				R	G	B	1.0					
R8G8B8X8_UNORM				R	G	B	1.0					
R8G8B8X8_UNORM_SRGB				R	G	B	1.0					
R9G9B9E5_SHAREDEXP				R	G	B	1.0					
B10G10R10X2_UNORM				R	G	B	1.0					
L16A16_FLOAT				L	L	L	A					
B5G6R5_UNORM		Yes		R	G	B	1.0					
B5G6R5_UNORM_SRGB				R	G	B	1.0					
B5G5R5A1_UNORM		Yes		R	G	B	A					
B5G5R5A1_UNORM_SRGB				R	G	B	A					
B4G4R4A4_UNORM		Yes		R	G	B	A					

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
B4G4R4A4_UNORM_SRGB				R	G	B	A					
R8G8_UNORM			DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R8G8_SNORM		Yes	DX10/OGL	R	G	0.0	1.0	DX9	R	G	1.0	1.0
R8G8_SINT			DX10/OGL	R	G	0.0	1.0					
R8G8_UINT			DX10/OGL	R	G	0.0	1.0					
R16_UNORM	Yes	Yes	DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_SNORM			DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_SINT			DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_UINT			DX10/OGL	R	0.0	0.0	1.0	OCL				
R16_FLOAT	Yes		DX10/OGL	R	0.0	0.0	1.0	OCL	R	1.0	1.0	1.0
I16_UNORM				I	I	I	I					
L16_UNORM				L	L	L	1.0					
A16_UNORM				0.0	0.0	0.0	A					
L8A8_UNORM		Yes		L	L	L	A					
I16_FLOAT				I	I	I	I					
L16_FLOAT				L	L	L	1.0					
A16_FLOAT				0.0	0.0	0.0	A					
L8A8_UNORM_SRGB				L	L	L	A					
R5G5_SNORM_B6_UNORM		Yes		R	G	B	1.0					
A1B5G5R5_UNORM				R	G	B	A					
A4B4G4R4_UNORM				R	G	B	A					
R8_UNORM		Yes	DX10/OGL	R	0.0	0.0	1.0					
R8_SNORM			DX10/OGL	R	0.0	0.0	1.0					
R8_SINT			DX10/OGL	R	0.0	0.0	1.0					
R8_UINT			DX10/OGL	R	0.0	0.0	1.0					
A8_UNORM		Yes		0.0	0.0	0.0	A					
I8_UNORM				I	I	I	I					
L8_UNORM		Yes		L	L	L	1.0					
Y8_UNORM		Yes		Y	Y	Y	1.0					
L8_UNORM_SRGB				L	L	L	1.0					
DXT1_RGB_SRGB				R	G	B	1.0					
R1_UNORM				R	0.0	0.0	1.0					
YCRCB_NORMAL		Yes		Cr	Y	Cb	1.0					
YCRCB_SWAPUVY		Yes		Cr	Y	Cb	1.0					
BC1_UNORM				R	G	B	A					

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
BC2_UNORM				R	G	B	A					
BC3_UNORM				R	G	B	A					
BC4_UNORM			DX10/OGL	R	0.0	0.0	1.0	OCL				
BC5_UNORM			DX10/OGL	R	G	0.0	1.0	OCL				
BC1_UNORM_SRGB				R	G	B	A					
BC2_UNORM_SRGB				R	G	B	A					
BC3_UNORM_SRGB				R	G	B	A					
YCRCB_SWAPUV				Cr	Y	Cb	1.0					
YCRCB_SWAPY				Cr	Y	Cb	1.0					
DXT1_RGB				R	G	B	1.0					
R8G8B8_UNORM				R	G	B	1.0					
R8G8B8_SNORM				R	G	B	1.0					
BC4_SNORM			DX10/OGL	R	0.0	0.0	1.0	OCL				
BC5_SNORM			DX10/OGL	R	G	0.0	1.0	OCL				
R16G16B16_FLOAT				R	G	B	1.0					
R16G16B16_UNORM				R	G	B	1.0					
R16G16B16_SNORM				R	G	B	1.0					
BC6H_SF16				R	G	B	1.0*					
BC7_UNORM				R	G	B	A					
BC7_UNORM_SRGB				R	G	B	A					
BC6H_UF16				R	G	B	1.0*					
PLANAR_420_8		Yes		Cr	Y	Cb	1.0					
PLANAR_420_16		Yes		Cr	Y	Cb	1.0					
R8G8B8_UNORM_SRGB				R	G	B	1.0					
ETC1_RGB8				R	G	B	1.0					
ETC2_RGB8				R	G	B	1.0					
EAC_R11				R	0.0	0.0	1.0	OCL				
EAC_RG11				R	G	0.0	1.0	OCL				
EAC_SIGNED_R11				R	0.0	0.0	1.0	OCL				
EAC_SIGNED_RG11				R	G	0.0	1.0	OCL				
ETC2_SRGB8				R	G	B	1.0					
R16G16B16_UINT			DX10/OGL	R	G	B	1.0					
R16G16B16_SINT			DX10/OGL	R	G	B	1.0					
ETC2_RGB8_PTA				R	G	B	A					
ETC2_SRGB8_PTA				R	G	B	A					

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
ETC2_EAC_RGBA8				R	G	B	A					
ETC2_EAC_SRGB8_A8				R	G	B	A					
R8G8B8_UINT			DX10/OGL	R	G	B	1.0					
R8G8B8_SINT			DX10/OGL	R	G	B	1.0					
L8A8_UINT			DX10/OGL	L	L	L	A					
L8A8_SINT			DX10/OGL	L	L	L	A					
L8_UINT			DX10/OGL	L	L	L	1.0					
L8_SINT			DX10/OGL	L	L	L	1.0					
I8_UINT			DX10/OGL	I	I	I	I					
I8_SINT			DX10/OGL	I	I	I	I					

Programming Note	
Context:	SURFACE_STATE
Any surface format not supported by the 3D sampler except Raw Buffer format will return undefined results. The Unsupported Raw Buffer format will be treated as RGBA8888 by default.	

Programming Note	
Context:	1D Surfaces
1D Surfaces must be defined as linear (not tiled).	

Programming Note	
Context:	SURFACE_STATE
Surface formats PLANAR_420_8 and PLANAR_420_16 which require half-pitch chroma planes (e.g. YV12) cannot support fenced tiling.	

Programming Note	
Context:	Chroma Key and sample_unorm
The surface formats PLANAR_420_16, R16_UNORM, R16G16B16A16_UNORM, and R16G16_UNORM are only supported for sample_unorm with Chroma Key enabled.	

It is recommended, for performance reasons, to never use any format of the type L*A*, I* or A*. Instead use R* or RG* in combination with Shader Channel Select.

Programming Note	
Context:	Anisotropic Filtering with 128bpt and DX9 Border Color Mode
Anisotropic Filtering of 128bpt surfaces, when Sampler State **Texture Border Color Mode** is set to DX9, is not supported	

Programming Note	
Context:	NULL Surfaces and Shader Channel Select
Is SURFTYPE_NULL is selected, **Shader Channel Select Alpha** must be programmed to SCS_ZERO.	

Programming Note	
Context:	HiZ Surfaces
HiZ Auxiliary surfaces are not supported for surfaces which are sampled by the 3D sampler. If an Auxiliary surface is programmed into the RENDER_SURFACE_STATE it will be ignored by the 3D sampler and interpreted as AUX_NONE.	

Programming Note	
Context:	Re-described surfaces
If two surface states reference the same texture with different **Surface Format** fields there is a possibility that the sampler will return incorrect pixels unless a texture cache invalidate is done between accesses using different surface state.	

Programming Note	
Context:	Quilted Surfaces
Quilted Surfaces are not supported.	

Programming Note	
Context:	Planar Surfaces with Disabled Sampler
A sample to a YUV Planar surface (e.g. PLANAR_420_8) when the Sampler State has been disabled by setting the Sampler Disable bit, will not return 0's.	

Programming Note	
Context:	R10G10B10_FLOAT_A2_UNORM Point-Sample Precision
When a sampler returns a point-sample to a R10G10B10_FLOAT_A2_NORM surface and the return format is 32-bit Floating Point, the Alpha channel will only have the precision of a 16-bit floating point value (10-bits of fractional precision).	

Programming Note	
Context:	Intensity, Luminance and Alpha
Intensity, Luminance and Alpha surface formats which support Shadow Mapping (e.g. L16_UNORM) must be bound as the equivalent R/G/B surface (e.g. R16_UNORM) with shader channel selects programmed to return the correct value in each channel. The list of these formats is below: I24X8_UNORM L24X8_UNORM	

Programming Note	
Context:	Intensity, Luminance and Alpha
A24X8_UNORM I32_FLOAT L32_FLOAT A32_FLOAT I16_UNORM L16_UNORM A16_UNORM I16_FLOAT L16_FLOAT A16_FLOAT See the section on Sampler SW Performance Hints for a complete table of how to map I/L/A formats to R/G/B with the correct Shader Channel Selects	

Programming Note	
Context:	Sample Tap Discard and YUV Surfaces
Returned Filter Weight for Sample Tap Discard will only report the filter weight contributing texel values of the Lumina plane for YUV surface such as YCrCb.*	

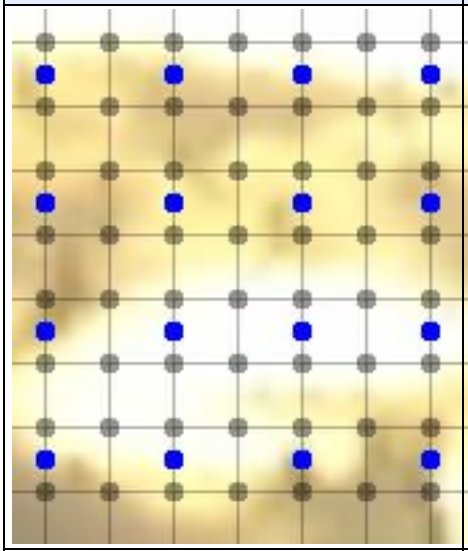
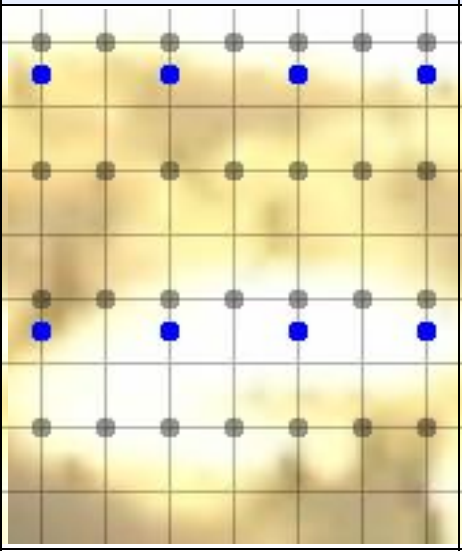
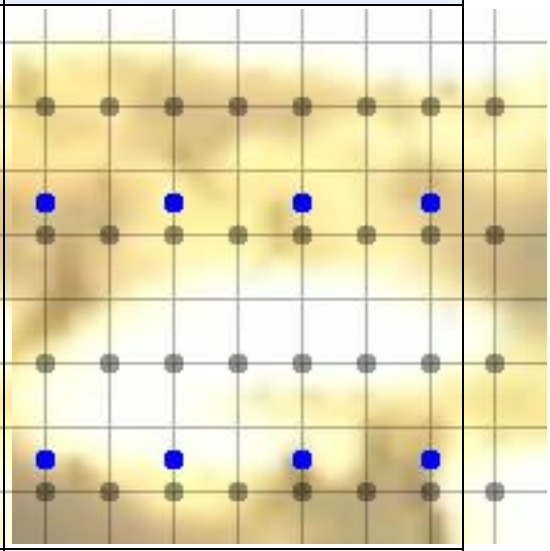
Programming Note	
Context:	Corner Texel Mode with Large Texel Addresses
When Corner Texel Mode is enabled for a surface and the resulting texel address of a sample is greater than 2^18, the texel address will be truncated by 1ULP which may cause image corruption. This corruption is generally visible for cases which utilize TEXCOORDMODE_CLAMP, CLAMP_BORDER, and HALF_BORDER	

SURFACE_STATE for Deinterlace sample_8x8

This section contains media surface state definitions.

Cr(V)/Cb(U) Pixel Offset V Direction

The position of Y is brown and the position of Cr(V)/Cb(U) is blue.

Full Frame	Top Field	Bottom Field
		
V Offset 0.5	V Offset 0.25	V Offset 0.75

MEDIA_SURFACE_STATE

Programming Note	
Context:	SURFACE_STATE for Deinterlace sample_8x
The Faulting modes described in the MEMORY_OBJECT_CONTROL_STATE should be set to the same for the multi-surface Video Analytics functions like "LBP Correlation" and "Correlation Search" for both the surfaces.	

SAMPLER_STATE

SAMPLER_STATE has different formats, depending on the message type used. The sample_8x8 and deinterlace messages use a different format of SAMPLER_STATE as detailed in the corresponding sections.

Restriction: The **Min LOD** and **Max LOD** fields need range increased from [0.0,13.0] to [0.0,14.0] and fractional bits increased from six to eight. This requires a few fields to be moved as indicated in the text.

All 16-byte sampler states in a 64-byte aligned cache-line must be programmed with a valid sampler state.

Chroma Key cannot be used in conjunction with Anisotropic filtering.

SAMPLER STATE

SAMPLER_BORDER_COLOR_STATE

The 3D Sampler only supports 32-bit per channel border color formats, which are then converted by HW to the appropriate surface format. The border color is NOT used for missing channels (e.g., the Blue Channel in R16G16_FLOAT).

The default behavior is to return 0 for missing channels, but this can be changed to return 1 for missing channels by using the Shader Channel Selects in the RENDER_SURFACE_STATE.

Sampler State
3DSTATE_CHROMA_KEY
SAMPLER_INDIRECT_STATE

Programming Note	
Context:	Wrap with very small negative coordinates
If the input coordinate is negative and very small (< 2^-25) and the TEXCOORDMODE_WRAP is selected for that coordinate, the sampler will truncate the texel coordinate to 0.0 and return the wrong texel.	

Programming Note	
Context:	Chroma Key
Chroma Key is not supported. The **Chroma Key Enable** field in the SAMPLER_STATE will be ignored.	

Messages

The sampler receives messages from shader clients. These messages contain information to allow the sampler to perform sample operations and return results. A message consists of four components:

- Execution Mask: Indicates, for a given SIMD, which pixels are valid.
- Message Descriptor: Required information including length of the message, and the length of the response.
- Message Header: Optional information which may be required for certain operations (e.g. Direct Write to Render Target).
- Message Payload: Specific data for each sampler operation including coordinates and other message parameters.

Messages from the EU can be "fused", meaning that a pair of threads will simultaneously issue identical sample operations (for different pixels) to a single 3D sampler. The 3D sampler will process these two messages in parallel. Two fused SIMD8 messages effectively become SIMD16 and two fused SIMD16 messages effectively become a SIMD32 message. It is required that the two messages utilize the exact same state (surface and sampler) because only one state will be fetched for both messages. In addition, if a header is present, it must be present on both messages and the contents of the header must be identical. See the section on Fused Send Message Handling for additional details.

Programming Notes

Programming Note	
Context:	Shader restrictions
Messages which require the calculation of LOD in the sampler must send pixels coordinates for each group of 4 pixels (known as a sub-span) even if all pixels are not enabled for return to the shader. The LOD calculation and the subsequent return of pixels will not be correct if all pixels in a sub-span do not have valid coordinates or gradients to allow calculation of LOD.	

Programming Note	
Context:	3D Sampler Messages
Write back messages can return erroneous information in the Pixel Null Mask field for sample_c messages on a 64-bit surface format (e.g. R16G16B16A16_UNORM)	

Programming Note	
Context:	Message Headers and Mid-Thread Pre-Emption
A message header is required for GPGPU kernels for all sample_8x8 and sample_unorm messages to allow save/restore mechanisms to work correctly.	

Procedural Textures and Texel Shading

Evaluate Messages

The 3D sampler supports usage by Pixel Shaders which utilize the evaluate() type messages for Procedural Textures. The format of these messages to sampler are identical to normal sample() messages except in two respects:

- The return length of the message (as defined in the Sampler Message Descriptor) must be set to zero to indicate this is an evaluate operation
- The RENDER_SURFACE_STATE **Procedural Texture** bit field must be set.

If both of the above conditions hold for an incoming sampler message, the 3D sampler will interpret them as an evaluate message, and will not perform the sample operation, but will send the requested texels to the AMFS block to spawn the required Texel Shaders.

The table below shows the behavior of 3D sampler for incoming messages for all possible states of message return length and **Procedural Texture** field in RENDER_SURFACE_STATE.

Return Length	Procedural Texture state bit	Behavior
0	0	Message Will be Dropped, this is an unsupported operation
0	1	Message is interpreted as an evaluate message and all required texels will be sent to AMFS for spawning of a Texel Shader
>0	0	Legacy Behavior: Sampler will perform sample operations as normal and return sample result to shader
>0	1	Sampler will perform sample operations as normal and return sample result to shader

Restrictions

Programming Note	
Context:	Tiling
Procedural Textures and associated evaluate messages to 3D sampler are only supported for TileYs surfaces.	

Programming Note	
Context:	Bindless Surfaces
Procedural Textures and associated evaluate messages to 3D sampler are only supported for Bindless Surface and Bindless Sampler State	

Programming Note	
Context:	Non-Eval Messages
AMFS Evaluation Shaders must not use the same surface state as a Non-Evaluation Shader to the same surface.	

Each 4K page fits in a 16byte value and within the 16 bytes, the below math will help get us to the right nibble –

Tile Format	SURF_TYPE	Byte selector	Nibble selector
Tile4	2D,3D,CUBE	Main_Surface_Adr[13, 12, 11, 10, 9, 8]	Main_Surface_Adr[7]
Tile64	2D,3D,CUBE	Main_Surface_Adr[13, 12, 11, 10, 9, 8]	Main_Surface_Adr[7]

Programming Note	
Context:	non-compute shaders doing evaluate
SW has to send a CS stalling flush before issuing AMFS flush to the 3D pipe. All shaders that perform evaluates must send a Cache Flush message to the sampler with a non-zero read-length after all evaluates are issued and before End-Of-Thread	

Programming Note	
Context:	Evaluate in compute context
1. A pipe control flush with "AMFS flush Enable" set and "DC flush enable set" must be sent down the pipe before a context switch, when compute shaders do evaluate. 2. if compute shader evaluates, and SW needs to flush the AMFS pipe, it has to first send a pipecontrol flush to the compute pipe and then switch to 3D pipe before sending a pipecontrol with "Command Streamer Stall Enable", AMFS flush Enable, and DC flush enable set on it	

Programming Note	
Context:	Evaluate in compute context & pre-emption
1) If shaders do evaluate (3d/compute), pre-emption must be disabled, until AMFS data is flushed out of all the caches. 2) All shaders that perform evaluates must send a Cache Flush message to the sampler with a non-zero read-length after all evaluates are issued and before End-Of-Thread	

Programming Note	
Context:	Evaluate in compute context
Compute shaders run on a CCS context must not issue AMFS evaluates. All AMFS evaluates must run in an RCS context	

Workaround	
Context:	3D and compute
Any kernel that contains AMFS evaluate (WriteSamplerFeedback) operations must issue two back-to-back sampler cache flush messages after all evaluate operations are sent and before the kernel EOT message. 1. The first sampler cache flush message must have a zero-length return. This is used to signal EOT to the AMFS unit. 2. The second sampler cache flush message must have a non-zero-length return. This is used to block the kernel EOT until all AMFS operations are flushed out of the sampler. Failure to do both sampler cache flush messages can result in HW hangs and/or spurious page faults.	

Pairing bit is always bit 6

Message Descriptor and Execution Mask

Execution Mask

SIMD16. The 16-bit execution mask forms the valid pixel signals. This determines which pixels are sampled and results returned to the GRF registers. Samples for invalid pixels are not overwritten in the GRF. However, if LOD needs to be computed for a subspan based on the message type and MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, as these are needed for the LOD computation.

SIMD8. The low 8 bits of the execution mask form the valid pixel signals. If LOD needs to be computed based on MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, since these are needed for the LOD computation.

SIMD32. The execution mask is ignored, all pixels are considered valid, and all channels are returned regardless of the execution mask.

Message Descriptor

The Sampler Message Descriptor is composed of a 32-bit Message Descriptor and a 32-bit Extended Message Descriptor (used for support Bindless resources, CPS etc.) which is dedicated to the 26-bit Bindless Surface Offset field for Bindless surfaces. For Non-Bindless surfaces the lower 11-bits of the extended descriptor are used by EU.

The message descriptor is sent in parallel with the message payload (and header if used).

Descriptor
Message Descriptor - Sampling Engine
Non-Bindless Extended Message Descriptor - Sampling Engine
Bindless Extended Message Descriptor - Sampling Engine
Extended Message Descriptor - Execution Unit

Programming Note	
Context:	3D Sampler Messages

Restrictions for Message Descriptors

Use of any message to the Sampling Engine function with the **End of Thread** bit set in the message descriptor is not allowed.

Message Header

The message header for the sampling engine is the same regardless of the message type. The message header is optional. If the header is not present, the behavior is as if the message was sent with all fields in the header set to zero and the write channel masks are all enabled and offsets are zero. However, if the

header is not included in the message, the Sampler State Pointer will be obtained from the command stream input for the given thread.

For fused messages, the two messages must contain the same content in the message header or both messages must not have a header.

The exact definition of the Sampler Message Header can be found in the section below:

Sampler Message Header

Message Types

3D Sampler Message Types

The 3D sampler supports multiple message types with different types of behaviors being supported. Each message can be supported with multiple SIMD forms (e.g. SIMD8, SIMD16 etc). See the section Message Formats for which SIMD forms are supported as well as the specific parameters and order of parameters for each message.

Below is a complete list of supported 3D Sampler message types:

Message Types
sample_*
ld_*
gather4_*
LOD
sampleinfo
resinfo
cache_flush

Common Message Variants

Many message types have multiple variants which provide for different sampling behaviors. The variants of a message type are named by appending a suffix to the base message.

Variant Suffix	Definition
_l	LOD Override: The LOD is provided in the message rather than being calculated from the gradients of the sub-span pixel coordinates.
_b	LOD Bias: A floating-point value between +16.0 and -16.0 is added to the LOD based on gradients of the sub-span pixel coordinates.
_c	Compare: Returns a white or black result depending on the comparison of a Ref parameter to the resulting red-channel of the sample. Comparison type is defined by the **Shadow Function** field in the SAMPLER_STATE
_lz	LOD=0 Override: LOD is forced to 0. No LOD is calculated or provided in the message
_d	Gradient: Rather than receiving absolute texel coordinates for all pixels of sub-span, a single pixel coordinate tuple is provided and a set of floating-point gradient values with respect to those pixel

Variant Suffix	Definition
	coordinates. This is then used to calculate other pixel coordinates and the LOD
_killpix	Kill Pixel: Used in conjunction with the Chroma Key mode enabled by the Sampler State Field CHROMA_KEY_ENABLE. It causes the associated sampler result to include a "Kill Pixel Mask" where 0's indicate pixels which matched a particular Chroma Key Mode.

See the subsections for each message type for a description of the behaviors and programming restrictions.

Restrictions and Programming Notes for All Message Types

Programming Note	
Context:	Message Types
For surfaces of type SURFTYPE_CUBE, the sampling engine requires u, v, and r parameters that have already been divided by the absolute value of the parameter (u, v, or r) with the largest absolute value.	

Programming Note	
Context:	Reduction Filter and *_c message variants.
Programming restrictions defined for *_c variants of all message types will not apply if the reduction filter is enabled via the Reduction Type Enable field in the SAMPLER_STATE and a Reduction Type that is **NOT** COMPARISON is selected. The restrictions of the non *_c variant will apply instead.	

Programming Note	
Context:	Errata
very small float AI (<2.22045E-16) (that are not zero or denormal) may not convert correctly to integer in the case of 1D array and 2D array messages, can cause wrong AI use.	

Sample Message Types

Sample Message Definition

Message Type	Description
sample_*	The surface is sampled using the indicated sampler state. LOD is computed using calculated gradients between adjacent pixels. One, two, or three floating-point coordinate parameters may be specified depending on how many dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters are defaulted to 0. The sample message enables filtering/blending operations (e.g. MAP_LINEAR) and addressing modes (e.g. Mirror) which are controlled by SAMPLER_STATE.

Supported Variants

Message	Description
sample	Basic sample operation as described above.
sample_l	Basic sample with LOD in message, not computed
sample_b	Basic sample with Bias in messaged added to computed LOD.
sample_c	Basic sample with Reference value in message and comparison against Red channel of the sampled surface returned. Used primarily for shadow maps. Comparison type is defined by the **Shadow Function** field in the SAMPLER_STATE.
sample_lz	Basic sample with LOD forced to 0. Possibly higher performance than **sample_l** for cases where the surface has MIP_COUNT=1 because the LOD does not need to be computed or sent in message.
sample_l_c	sample_c with LOD override in message, not computed.
sample_b_c	sample_c with LOD Bias
sample_c_lz	Similar to sample_c with LOD forced to 0. Possibly higher performance than **sample_c** or **sample_l_c** for cases where the surface has MIP_COUNT=1 because the LOD does not need to be computed.
sample_d	The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The *r* coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the u, v, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The *r* coordinate contains the faceid, and the r gradients are ignored by hardware. Previously known as sample_g.
sample_d_c	Same as sample_d, but returns comparison against Red Channel of sampled surface like sample_c. Previously known as sample_g_c.
sample_killpix	Basic sample, but returns a Kill Pixel Mask based on Chroma Key sampler comparison results. An additional register is returned after the sample results which contains the kill pixel mask. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask.

Restrictions and Programming Notes for Sample

Programming Note
sample: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE.
sample: Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
sample: The *mlod* parameter specifies the per-pixel MinLOD.

Restrictions and Programming Notes for sample_b

Programming Note
sample_b: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. The LOD bias delivered in the **bias** parameter is restricted to a range of [-16.0, +16.0). Values outside this range produce undefined results.
sample_b: The *mlod* parameter specifies the per-pixel MinLOD.

Restrictions and Programming Notes for sample_l and sample_lz

Programming Note
sample_l and sample_lz: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
sample_l and sample_lz: The ld_lz message is the same as ld except the lod parameter is set to zero instead of being delivered.

Restrictions and Programming Notes for sample_c and sample_c_lz

Programming Note
sample_c and sample_c_lz: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE. The Surface Format of the associated surface must be indicated as supporting shadow mapping as indicated in the surface format table. MIPFILTER_LINEAR, MAPFILTER_LINEAR, MAPFILTER_ANISOTROPIC are allowed even for surface formats that are listed as not supporting filtering in the surface formats table. Use of the SIMD4x2 form of *sample_c* without **Force LOD to Zero** enabled in the message header is not allowed, as it is not possible for the hardware to compute LOD for SIMD4x2 messages. Using SURFTYPE_CUBE surfaces is undefined with the following surface formats: I24X8_UNORM, L24X8_UNORM, A24X8_UNORM, I32_FLOAT, L32_FLOAT, and A32_FLOAT. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1. Using DX9 **Texture Border Color Mode** and either of the following is undefined: • Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER. • Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.

Programming Note
sample_c and sample_c_lz: Must use the DX10 BORDER_COLOR_MODE (selected in sampler state) when using CLAMP_BORDER addressing mode.
sample_c: The *mlod* parameter specifies the per-pixel MinLOD.

Restrictions and Programming Notes for sample_l_c

Programming Note
sample_l_c: All restrictions applying to both sample_l and sample_c must be honored.
sample_l_c: Must use the DX10 BORDER_COLOR_MODE (selected in sampler state) when using CLAMP_BORDER addressing mode.

Restrictions and Programming Notes for sample_b_c

Programming Note
sample_b_c: All restrictions applying to both sample_bl and sample_c must be honored.
sample_b_c: All variations of the sample_c, must use the DX10 BORDER_COLOR_MODE (selected in sampler state) when using CLAMP_BORDER addressing mode.

Restrictions and Programming Notes for sample_d

Programming Note
sample_d: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
sample_d: The mlod parameter specifies the per-pixel MinLOD.

Restrictions and Programming Notes for sample_d_c

Programming Note
sample_d_c: All restrictions applying to both sample_d and sample_c must be honored.

Restrictions and Programming Notes for sample_killpix

Programming Note
sample_killpix: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. sample_killpix is supported only in SIMD8 mode. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.

Programming Note
sample_killpix: If the sampler is disable via the **Sampler Disable** bit in the SAMPLER_STATE, the Kill Pixel Mask returned will be undefined.

Restrictions and Programming Notes for sample_min, sample_max

Programming Note
The Surface Type of the associated surface must be SURFTYPE_2D and **Surface Array** must be disabled. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1
If sampling on a 3D surface (Volumetric),the following two conditions cannot be supported on a sample_min or sample_max at the same time: • Sampler state address mode for q direction specifies D3D11_TEXADDRESS_BORDER • Address mode for U and V do not specify D3D11_TEXADDRESS_BORDER
If Null and/or Border Texels are being excluded by using the Return Filter Weight fields in the SAMPLER_STATE, and the surface type includes an Alpha Channel, the Alpha channel must be obtained separately by having the Return Filter Weight fields cleared, and the resulting Alpha will still contain contributions from NULL or Border Texels.

Restrictions and Programming Notes for sample_lz

Programming Note
sample_lz: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D

ld Message Types

ld Message Definition

Message Type	Description
ld_*	The surface is sampled using a default sampler state. It performs a "point sample" with the LOD provided in the message rather than being calculated by gradients and the coordinates are non-normalized integers rather than normalized floating-point values. If the message doesn't include an LOD parameter, the message samples from LOD 0. For **ld_*** message types, the sampler state is defaulted as follows: • min, mag, and mip filter modes are "nearest". • All address control modes are "zero", a special mode in which any texel off the map or outside the MIP range of the surface has a value of zero in all channels, except for surface formats

Message Type	Description
	without an alpha channel, which return a value of one in the alpha channel. The ld* family of messages do not perform any blending operation. Additional parameters are needed for the variants of ld defined below. Potentially lower power than using **sample_l** with Filter Mode=MAP_NEAREST because coordinates do not need to be converted from floating-point.

Supported Variants

Message	Description
ld	Basic ld operation as described above.
ld_lz	Basic ld with LOD forced to 0. Reduces the size of the message with potential performance and power advantage because LOD is not sent as part of the message.
ld2dms_w	Same as ld2dms but provides more bits for MCS to support multi-sampled surfaces which are X16 MSAA. Therefore, the ld2dms_w message has two MCS parameters, named mcsl and mcsh. These parameters contain the low and high halves of the 64-bit MCS value, respectively. However, use of the ld2dms_w message is allowed with Number of Multisamples set to 2, 4, 8, or 16. With this message, if Number of Multisamples is 2, 4 or 8, the ld2dms_w message behaves the same as ld2dms, with mcsl used for mcs, and mcsh ignored. For SIMD8/SIMD16 the mcsh parameters is only used for X16 MSAA, and should be all 0's for X8 MSAA or lower depth.
ld_mcs	Basic ld operation, but used to fetch the MCS auxiliary surface data rather than pixels from the surface itself. The returned MCS data is used to construct the ld2dms_w message to sample from the multi-sampled surface

Restrictions and Programming Notes for ld

Programming Note
ld: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1

Restrictions and Programming Notes for ld_lz

Programming Note
ld_lz: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message.

Restrictions and Programming Notes for ld_mcs

Programming Note
ld_mcs: The Surface Type of the surface associated with the auxiliary surface must be SURFTYPE_2D.
ld_mcs: The ld_mcs message uses the Auxiliary Surface Base Address and Auxiliary Surface Pitch fields in SURFACE_STATE to determine the base address and pitch of the surface. Surface Format is overridden to R8_UINT if Number of Multisamples is 2 or 4, R32_UINT if Number of Multisamples is 8, or R32G32_UINT if Number of Multisamples is 16. This message cannot be used on a non-multisampled surface. Otherwise, ld_mcs behaves like the ld message. If ld_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_MCS, this message returns zeros in all channels.
ld_mcs: All LOD parameters must be the same value within a single message (i.e. all samples returned for message must be from a single MIP).
ld_mcs: If ld_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_CCS_D, this message returns the value which indicates that all planes are uncompressed. This value can be used directly in the ld2dms_w message to ensure the correct sub-pixels are fetched. ld_mcs always returns 512 bits of data for SIMD8 and 1024 bits of data for SIMD16 regardless of the MSAA depth of the surface. If ld_mcs is issued on an auxiliary surface associated with a X8 MSAA or lower depth, the upper half of the return data will be 0's.

Restrictions and Programming Notes for ld_2dms_w

Programming Note
ld_2dms_w: The Surface type must not be MULTISAMPLECOUNT_1. The Surface Type of the associated surface must be SURFTYPE_2D.
ld_2dms_w: The MCS parameter is ignored for UMS (uncompressed MSAA) surfaces which do not have an Auxiliary MCS surface. In this case, the ssi parameter provided in the message is used to determine which sub-pixel to fetch.

Restrictions and Programming Notes for ld

Programming Note
ld: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1

Restrictions and Programming Notes for ld_lz

Programming Note
ld_lz: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message.

Restrictions and Programming Notes for ld_mcs

Programming Note
ld_mcs: The Surface Type of the surface associated with the auxiliary surface must be SURFTYPE_2D.
ld_mcs: The ld_mcs message uses the Auxiliary Surface Base Address and Auxiliary Surface Pitch fields in SURFACE_STATE to determine the base address and pitch of the surface. Surface Format is overridden to R8_UINT if Number of Multisamples is 2 or 4, R32_UINT if Number of Multisamples is 8, or R32G32_UINT if Number of Multisamples is 16. This message cannot be used on a non-multisampled surface. Otherwise, ld_mcs behaves like the ld message. If ld_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_MCS, this message returns zeros in all channels.
ld_mcs: All LOD parameters must be the same value within a single message (i.e. all samples returned for message must be from a single MIP).
ld_mcs: If ld_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_CCS_D, this message returns the value which indicates that all planes are uncompressed. This value can be used directly in the ld2dms_w message to ensure the correct sub-pixels are fetched. ld_mcs always returns 512 bits of data for SIMD8 and 1024 bits of data for SIMD16 regardless of the MSAA depth of the surface. If ld_mcs is issued on an auxiliary surface associated with a X8 MSAA or lower depth, the upper half of the return data will be 0's.

Restrictions and Programming Notes for ld_2dms_w

Programming Note
ld_2dms_w: The Surface type must not be MULTISAMPLECOUNT_1. The Surface Type of the associated surface must be SURFTYPE_2D.
ld_2dms_w: The MCS parameter is ignored for UMS (uncompressed MSAA) surfaces which do not have an Auxiliary MCS surface. In this case, the ssi parameter provided in the message is used to determine which sub-pixel to fetch.

gather4 Message Types

Definitions

Message Type	Description
gather4_*	gather4* messages ignore min/mag/mip filter modes and instead does four point samples using (u,v) texture coordinate with deltas at the following locations: (-,+),(+,+),(+,-),(-,-), where the magnitude of the deltas are always half a texel. For gather4_i/b where there is an implicit LOD, it is calculated the same as mip_filter_nearest. For gather4, the LOD is forced to zero. The selected color channel the four contributing texels are returned directly in the sample's corresponding four channels as follows: upper left sample = alpha channel / upper right sample = blue channel lower left sample = red channel / lower right sample = green channel Returned color channel is selected through a message field: **Gather4 Source Channel Select** Two or three floating-point coordinates may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters default to 0.
gather4_*	If the Return Filter Weight fields in SAMPLER_STATE are set, any texel which is NULL and/or Border will be set to 0.0 in the returned result.

Supported Variants:

Message	Description
gather4	Basic gather4 behavior as described above.
gather4_c	Similar to basic gather4 but performs a compare between a Reference parameters and the gathered pixels and returns a white or black (1 or 0). In addition, like **sample_c**, it only returns data on the Red channel, so the **Gather4 Source Channel Select** must be set to Red.

Restrictions and Programming Notes for gather4:

Programming Note
gather4: The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
gather4: If Surface Format is R10G10B10_SNORM_A2_UNORM and Gather4 Source Channel Select is alpha channel, the returned value may be incorrect.
gather4: When using gather4 type messages on CUBEMAP surfaces with SINT* surface formats the ChromaKeyEnable state bit should be set and the ChromaKeyMode state set to KEYFILTER_KILL_ON_ANY_MATCH.

Restrictions and Programming Notes for gather4_c:

Programming Note
gather4_c: The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_CUBE. The Surface Format of the associated surface must be one of the following: R32_FLOAT_X8X24_TYPELESS, R32_FLOAT, R24_UNORM_X8_TYPELESS, or R16_UNORM. The channel selected by the Gather4 Source Channel Select field in the message header must be set to **Red**. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
gather4_c: When using gather4 type messages on CUBEMAP surfaces with SINT* surface formats the ChromaKeyEnable state bit should be set and the ChromaKeyMode state set to KEYFILTER_KILL_ON_ANY_MATCH.

sampleinfo Message Type

sampleinfo Message Definition

Message Type	Description
sampleinfo	The surface indicated in the surface state is not sampled. Instead, the number of samples (UINT32) and the sample position palette index (UINT32) for the surface are returned in the red and alpha channels respectively as UINT32 values. The sample position palette index returned in alpha is incremented by one from its value in the surface state. The **Sampler State Pointer** and **Sampler**

Message Type	Description
	Index are ignored. The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_NULL.

Supported Variants:

None

Restrictions and Programming Notes for sampleinfo:

Programming Note
sampleinfo: The return format for sampleinfo message must be 32-bits.

LOD Message Type

LOD Message Definition

Message Type	Description
LOD	The surface indicated in the surface state is not sampled. Instead, LOD is computed as if the surface will be sampled, using the indicated sampler state, and the clamped and unclamped LOD values are returned in the red and green channels, respectively, in FLOAT32 format. The blue and alpha channels are undefined, and can be masked to avoid returning them. LOD is computed using gradients between adjacent pixels. Three parameters are always specified, with extra parameters not needed for the surface being ignored.

Supported Variants:

None

Restrictions and Programming Notes for LOD:

Programming Note
LOD: The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
LOD:

Programming Note
The return format for LOD message must be 32-bits.

resinfo Message Type

resinfo Message Definition

Message Type	Description
resinfo	The surface indicated in the surface state is not sampled. Instead, the width, height, depth, and MIP count of the surface are returned as indicated in the table below. The format of the returned data is UINT32. In the case of a 1D or 2D surfaces, the depth value returned is the number of array slices. For non-arrayed 1D and 2D surfaces the value returned will be 1. The width, height, and depth may be scaled by the LOD parameter provided in the message to give the correct dimensions of the specified mip level. The LOD parameter is an unsigned 32-bit integer in this mode (note that negative values are out-of-range when interpreted as unsigned integers). The Sampler State Pointer and Sampler Index are always ignored. For SURFTYPE_1D, 2D, 3D, and CUBE surfaces, if the delivered LOD is outside of the range [0..MipCount-1], the returned values in the red, green, and blue channels are 0s.
resinfo	The surface indicated in the surface state is not sampled. Instead, the width, height, and depth of the surface are returned as indicated in the table below. The format of the returned data is UINT32. The Sampler State Pointer and Sampler Index are always ignored.

Returned Parameters for **resinfo:**

For the resinfo message the Red, Green and Blue channels returned contain the following specialized information.

Surface Type	Red	Green	Blue	Alpha
SURFTYPE_1D	(Width+1)»LOD	Surface Array ? Depth+1 : 0	0	MIP Count
SURFTYPE_2D	(Width+1)»LOD) * (QuiltWidth+1)	((Height+1)»LOD)* (QuiltHeight+1)	Surface Array ? Depth+1 : 0	MIP Count
SURFTYPE_3D	(Width+1)»LOD	(Height+1)»LOD	(Depth+1)»LOD	MIP Count
SURFTYPE_CUBE	(Width+1)»LOD	(Height+1)»LOD	Surface Array ? Depth+1 : 0	MIP Count
SURFTYPE_BUFFER SURFTYPE_SCRATCH	Buffer size (from combined Depth/Height/Width) If buffer size is exactly 2^32, zero is returned in this field.	Undefined	Undefined	Undefined
SURFTYPE_NULL	0	0	0	0

Supported Variants:

None

Restrictions and Programming Notes for resinfo:

Programming Note
resinfo: The return format of resinfo message must be 32-bits.

cache_flush Message Type

cache_flush Message Definition

Message Type	Description
cache_flush	The texture caches in the sampling engine are invalidated. This includes all levels of texture cache, however the state caches are not invalidated. Any outstanding sample operations which arrive at the texture sampler prior to the cache_flush message are completed normally. Any sample operations which arrive after the cache_flush message will cause texture cache-misses and sampler will fetch textures from memory.

Supported Variants:

None

Restrictions and Programming Notes for cache_flush:

Programming Note
cache_flush: Software must ensure that this message is ordered appropriately with other messages. Hardware ensures only that this message is processed in the same order as the messages are received. The message causes a write back message to indicate that the flush has been completed.
cache_flush: The cache_flush message must be sent at the end of any AMFS first-pass shader where the last instruction is not a send to sampler.

Message Format

The following section describes how messages are formatted, including how message length and response length are calculated. The sampler supports a variety of different message types which perform different sampling and filtering operations (see Message Types section for a description of these sampler and filtering operations). For each message type, different SIMD modes are supported which determines

how many pixels the sampler performs the specified operation on in parallel. The precision of the input parameters and resulting samples can also be controlled by programming the **Return Format** bit in the message descriptor (for sample precision) and using SIMD8H/SIMD8 and SIMD16H/SIMD16 message types (for coordinate precision).

Supported SIMD Types

The table below describes the SIMD modes which are supported. SIMD32 and SIMD64 are used for media-type operations only.

List of Supported Message SIMD Types

SIMD Modes	Description
SIMD8	8 samples with 32-bit Coordinates
SIMD16	16 samples with 32-bit coordinates
SIMD8H	8 samples with 16-bit Coordinates
SIMD16H	16 samples with 16-bit coordinates

Message Length

The **SIMD Mode** field determines the number of instances (i.e. pixels) to be sampled and the formatting of messages. The **Message Length** field indicates the number of parameters sent with the message. Higher-numbered parameters are optional, and default to a value of 0 if not sent but needed for the surface being sampled. The **Header Present** field determines whether a header is delivered as the first phase of the message or the default header from R0 of the thread's dispatch is used. A header will increase the length of the message by 1. Note that messages with more than 4 parameters will perform lower due to the additional information being sent to the sampler than messages with 4 or fewer parameters. For example, a sample message using 5 parameters will not be able to sustain the same throughput as a sample message with only 4 valid parameters.

Programming Note	
Context:	
All messages except sampleinfo or cache_flush that had a header must have at least one parameter in addition to the header.	

Response Length

The Response Length field determines the number of 256-bit registers which are used to receive the result of each message type for each SIMD mode.

RC = number of Return Channels as indicated by the **Write Channel Mask** bits.

SIMD Mode	Return Precision (bits per channel)	Response Length
SIMD8/SIMD8H	32	RC
SIMD8/SIMD8H	16	ceil(RC/2)

SIMD Mode	Return Precision (bits per channel)	Response Length
SIMD16/SIMD16H	32	RC
SIMD16/SIMD16H	16	ceil(RC/2)

If Header Present is *disabled*, Response Lengths set to values in the table below for SIMD8, SIMD8H, SIMD16H, and SIMD16 messages will set the Write Channel Mask fields that are used by the hardware according to the table below. Response Length values not indicated in the table are not valid.

SIMD Mode	Return Format	Response Length	Alpha Write Channel Mask	Blue Write Channel Mask	Green Write Channel Mask	Red Write Channel Mask
SIMD8/ SIMD8H	32-bit/ 16-bit	1	1	1	1	0
	32-bit/ 16-bit	2	1	1	0	0
	32-bit/ 16-bit	3	1	0	0	0
	32-bit/ 16-bit	4	0	0	0	0
SIMD16/ SIMD16H	32-bit	2	1	1	1	0
	16-bit	1	1	1	1	0
	32-bit	4	1	1	0	0
	16-bit	2	1	1	0	0
	32-bit	6	1	0	0	0
	16-bit	3	1	0	0	0
	16-bit	4	0	0	0	0
	32-bit	8	0	0	0	0

If **Pixel Fault Mask Enable** is enabled, response length must be set to a value one larger than that indicated in the tables above.

Programming Note	
Context:	3D Sampler Messages - Message Format
Parameter 0 is required.	

Message Formats

The table below describes the format of each message type for every supported SIMD type.

The Message Type field in the message descriptor in combination with the Message Length to determine which message is being sent. The table defines all of the parameters sent for each message type. The

position of the parameters in the payload is given in the section following corresponding to the *SIMD mode* given in the table. The Mnemonic column specifies the DX10-equivalent shader instructions expected to be translated to each message type.

Message parameters are typically formed in sequential 256-bit registers and sent in order to the sampler to form a complete message. All parameters are of type IEEE_Float, except those in the The ld*, resinfo, and the offu, offv of the gather4_po[_c] instruction message types, which are of type signed integer. Any parameter indicated with a blank entry in the table is unused. A message register containing only unused parameters is not included as part of the message.

The table below shows the supported messages and their formats for SIMD8, SIMD16, SIMD8H and SIMD16H:

Encoding	Mnemonic	Parameters								Comment
		0	1	2	3	4	5	6	7	
00000	sample	u	v	r	ai	mlod				
00001	sample_b	bias	u	v	r	ai	mlod			SIMD8H and SIMD16H Only
00010	sample_l	lod	u	v	r	ai				SIMD8H and SIMD16H Only
00001	sample_b	bias_ai	u	v	r	mlod				SIMD8 and SIMD16 Only
00010	sample_l	lod_ai	u	v	r					SIMD8 and SIMD16 Only
00011	sample_c	ref	u	v	r	ai	mlod			
00100	sample_d	u	dudx	dudy	u	dvdx	dvdy	r	mlod	
00101	sample_b_c	ref	bias	u	v	r	ai			SIMD8H and SIMD16H Only
00110	sample_l_c	ref	lod	u	v	r	ai			SIMD8H and SIMD16H Only
00101	sample_b_c	ref	bias_ai	u	v	r				SIMD8 and SIMD16 Only
00110	sample_l_c	ref	lod_ai	u	v	r				SIMD8 and SIMD16 Only
00111	ld	u	v	lod	r					SIMD8 and SIMD16 Only
01000	gather4	u	v	r	ai					
01001	LOD	u	v	r	ai					
01010	resinfo	lod								
01011	sampleinfo									

Encoding	Mnemonic	Parameters								Comment
01100	sample+killpix	u	v	r						
01101	Reserved									
01110	Reserved									
01111	Reserved									
10000	gather4_c	ref	u	v	r	ai				
10100	sample_d_c	ref	u	dudx	dudv	v	dvdx	dvdy	r	
11000	sample_lz	u	v	r	ai					
11001	sample_c_lz	ref	u	v	r	ai				
11010	ld_lz	u	v	r						SIMD8 and SIMD16 Only
11100	ld2dms_w	si	mcs0	mcs1	mcs2	mcs3	u	v	r	SIMD8H and SIMD16H Only
11101	ld_mcs	u	v	r						SIMD8H and SIMD16H Only
11111	cache_flush									

Programming Note	
Context:	3D Sampler Messages - Message Format
The ld must not be used with SIMD8H SIMD16H which are for 16-bit floating-point operands.	

Programming Note	
Context:	lod_ai and bias_ai on SIMD16 and SIMD8

When doing cube arrays on sample_l and sample_b with SIMD8 or SIMD16 the AI parameters is combined with the LOD/bias parameter on the 9 LSBs.

pseudo code:

intAi = Ftouint rnde(ai)

intAiLsb &= 0x1FF

(f0) cmp intAi, intAilsb

intAiLSb = (f0) Sel intAiLsb, 511

lod_Ai = LOD & 0xFFFFFE00

lod_Ai |= intAiLSb

Note that this is NOT done for SIMD16H or SIMD8h. Also note that AI is only used on cube arrays.

Programming Note	
Context:	Cube maps and 3D for sample_d*

Sample_d* no longer supports the 3rd set of gradient so they must be converted to a 4 sample* messages. Note that if not doing course pixel shading it is also possible to convert to one sample_l message instead using the resinfo message and doing the LOD calculation in the shader.

Example for conversion to sample* message

original sample_d SIMD8 message.

U0	U1	U2	U3	U4	U5	U6	U7
DU/DX0	DU/DX1	DU/DX2	DU/DX3	DU/DX4	DU/DX5	DU/DX6	DU/DX7
DU/DY0	DU/DY1	DU/DY2	DU/DY3	DU/DY4	DU/DY5	DU/DY6	DU/DY7
V0	V1	V2	V3	V4	V5	V6	V7
DV/DX0	DV/DX1	DV/DX2	DV/DX3	DV/DX4	DV/DX5	DV/DX6	DV/DX7
DV/DY0	DV/DY1	DV/DY2	DV/DY3	DV/DY4	DV/DY5	DV/DY6	DV/DY7
R0	R1	R2	R3	R4	R5	R6	R7
DR/DX0	DR/DX1	DR/DX2	DR/DX3	DR/DX4	DR/DX5	DR/DX6	DR/DX7
DR/DY0	DR/DY1	DR/DY2	DR/DY3	DR/DY4	DR/DY5	DR/DY6	DR/DY7
AI0	AI1	AI2	AI3	AI4	AI5	AI6	AI7

change to 4 sample SIMD8 messages with 2 pixels each. Does not matter which pixels are assigned to which message.

U0	U0 + Du/Dx0	U0 + Du/Dy0	-	U4	U4 + Du/Dx4	U4 + Du/Dy4	-
V0	V0 + Dv/Dx0	V0 + Dv/Dy0	-	V4	V4 + Dv/Dx4	V4 + Dv/Dy4	-
R0	R0 + Dr/Dx0	R0 + Dr/Dy0	-	R4	R4 + Dr/Dx4	R4 + Dr/Dy4	-
AI0				AI4			

repeat for the other 6 pixels.

Parameter Types

sample*, LOD, and gather4 messages

For all of the sample*, LOD, and gather4 message types, all parameters are 32-bit or 16-bit floating point, except the 'mcs', 'offu', and 'offv' parameters. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Normalized values range from [0,1] across the surface, with values outside the surface behaving as specified by the **Address Control Mode** in that dimension. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension, with values outside the surface being clamped to the surface.

Surface Type	u	v	r	ai
SURFTYPE1D	normalized 'x' coordinate	unnormalized array index	ignored	ignored
SURFTYPE_2D	normalized 'x' coordinate	normalized 'y' coordinate	unnormalized array index	ignored
SURFTYPE_3D	normalized 'x' coordinate	normalized 'y' coordinate	normalized 'z' coordinate	ignored
SURFTYPE_CUBE	normalized 'x' coordinate	normalized 'y' coordinate	normalized 'z' coordinate	unnormalized array index

Ld* messages

For the ld message types, all parameters are 32-bit or 16-bit unsigned integers (Ld must be 32-bit), except the 'mcs' parameter. Usage of the u, v, and r parameters is as follows based on **Surface Type**. Unnormalized values range from [0,n-1] across the surface, where n is the size of the surface in that dimension. Input of any value outside of the range returns zero.

Surface Type	u	v	r
SURFTYPE1D	unnormalized 'x' coordinate	unnormalized array index	ignored
SURFTYPE_2D	unnormalized 'x' coordinate	unnormalized 'y' coordinate	unnormalized array index
SURFTYPE_3D	unnormalized 'x' coordinate	unnormalized 'y' coordinate	unnormalized 'z' coordinate
SURFTYPE_BUFFER	unnormalized 'x' coordinate	ignored	ignored

SIMD Payloads

This section contains the SIMD payload definitions.

SIMD16 Payload

The payload of a SIMD16 message provides addresses for the sampling engine to process 16 entities (examples of an entity are vertex and pixel). The number of parameters required to sample the surface depends on the state that the sampler/surface is in. Each parameter takes two message registers, with 8 entities, each a 32-bit floating point value, being placed in each register. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this. For example, a 2D map using "sample_b" needs only u, v, and bias, but must send the r parameter as well.

DWord	Bits	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in the section. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2.7	31:0	**Subspan 3, Pixel 3 (lower right) Parameter 0**
M2.6	31:0	**Subspan 3, Pixel 2 (lower left) Parameter 0**
M2.5	31:0	**Subspan 3, Pixel 1 (upper right) Parameter 0**
M2.4	31:0	**Subspan 3, Pixel 0 (upper left) Parameter 0**
M2.3	31:0	**Subspan 2, Pixel 3 (lower right) Parameter 0**
M2.2	31:0	**Subspan 2, Pixel 2 (lower left) Parameter 0**
M2.1	31:0	**Subspan 2, Pixel 1 (upper right) Parameter 0**
M2.0	31:0	**Subspan 2, Pixel 0 (upper left) Parameter 0**
M3 - Mn		Repeat packets 1 and 2 to cover all required parameters.

SIMD8 Payload

This message is intended to be used in a SIMD8 thread, or in pairs from a SIMD16 thread. Each message contains sample requests for just 8 pixels.

DWord	Bits	Description
M1.7	31:0	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in the section. Format = IEEE Float for all sample* message types, U32 for ld and resinfo message types.
M1.6	31:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.5	31:0	**Subspan 1, Pixel 1 (upper right) Parameter 0**
M1.4	31:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.3	31:0	**Subspan 0, Pixel 3 (lower right) Parameter 0**
M1.2	31:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.1	31:0	**Subspan 0, Pixel 1 (upper right) Parameter 0**
M1.0	31:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2 - Mn		Repeat packet 1 to cover all required parameters.

SIMD16H Payload

The payload of a SIMD16H message provides addresses for the sampling engine to process 16 entities (examples of an entity are vertex and pixel). The number of parameters required to sample the surface depends on the state that the sampler/surface is in. Each parameter takes one message register, with 16 entities, each a 16-bit floating point or integer value, being placed in each register. Each parameter always takes a consistent position in the input payload. The length field can be used to send a shorter message, but intermediate parameters cannot be skipped as there is no way to signal this.

DWord	Bits	Description
M1.7	31:16	**Subspan 3, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in the section. Format = IEEE Half Float for all sample* message types, U16 for ld and resinfo message types.
	15:0	**Subspan 3, Pixel 2 (lower left) Parameter 0**

DWord	Bits	Description
M1.6	31:16	**Subspan 3, Pixel 1 (upper right) Parameter 0**
	15:0	**Subspan 3, Pixel 0 (upper left) Parameter 0**
M1.5	31:16	**Subspan 2, Pixel 3 (lower right) Parameter 0**
	15:0	**Subspan 2, Pixel 2 (lower left) Parameter 0**
M1.4	31:16	**Subspan 2, Pixel 1 (upper right) Parameter 0**
	15:0	**Subspan 2, Pixel 0 (upper left) Parameter 0**
M1.3	31:16	**Subspan 1, Pixel 3 (lower right) Parameter 0**
	15:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.2	31:16	**Subspan 1, Pixel 1 (upper right) Parameter 0**
	15:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.1	31:16	**Subspan 0, Pixel 3 (lower right) Parameter 0**
	15:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.0	31:16	**Subspan 0, Pixel 1 (upper right) Parameter 0**
	15:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2 - Mn		Repeat packets 1 and 2 to cover all required parameters.

SIMD8H Payload

This message is intended to be used in a SIMD8 thread, or in pairs from a SIMD16 thread. Each message contains sample requests for just 8 pixels.

DWord	Bits	Description
M1.7:4		**Reserved**
M1.3	31:16	**Subspan 1, Pixel 3 (lower right) Parameter 0** Specifies the value of the pixel's parameter 0. The actual parameter that maps to parameter 0 is given in the table in the section. Format = IEEE Half Float for all sample* message types, U16 for ld and resinfo message types.
	15:0	**Subspan 1, Pixel 2 (lower left) Parameter 0**
M1.2	31:16	**Subspan 1, Pixel 1 (upper right) Parameter 0**
	15:0	**Subspan 1, Pixel 0 (upper left) Parameter 0**
M1.1	31:16	**Subspan 0, Pixel 3 (lower right) Parameter 0**
	15:0	**Subspan 0, Pixel 2 (lower left) Parameter 0**
M1.0	31:16	**Subspan 0, Pixel 1 (upper right) Parameter 0**
	15:0	**Subspan 0, Pixel 0 (upper left) Parameter 0**
M2 - Mn		Repeat packet 1 to cover all required parameters.

Writeback Message

Corresponding to the four input message definitions are four writeback messages. Each input message generates a corresponding writeback message of the same type .

Programming Note	
Context:	3D Sampler Writeback Message
The Pixel Null Mask field, when enabled via the **Pixel Null Mask Enable** will be incorrect for sample_c when applied to a surface with 64-bit per texel format such as R16G16BA16_UNORM. Pixel Null mask Enable may incorrectly report pixels as referencing a Null surface.	

Programming Note	
Context:	Returning Filter Weights
When the **Return Filter Weight for Border Texels** or the **Return Filter Weight for Null Texels** bits in Sampler Message Descriptor are set, the 3D Sampler will return a filter weight in the Alpha Channel.	

SIMD16

Return Format = 32-bit

A SIMD16 writeback message consists of up to 8 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0 and regid+1, and alpha to regid+2 and regid+3. The pixels written within each destination register is determined by the execution mask on the "send" instruction.

If Pixel Null Mask Enable is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bit	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**

DWord	Bit	Description
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1.7	31:0	**Subspan 3, Pixel 3 (lower right) Red**
W1.6	31:0	**Subspan 3, Pixel 2 (lower left) Red**
W1.5	31:0	**Subspan 3, Pixel 1 (upper right) Red**
W1.4	31:0	**Supspan 3, Pixel 0 (upper left) Red**
W1.3	31:0	**Subspan 2, Pixel 3 (lower right) Red**
W1.2	31:0	**Subspan 2, Pixel 2 (lower left) Red**
W1.1	31:0	**Subspan 2, Pixel 1 (upper right) Red**
W1.0	31:0	**Supspan 2, Pixel 0 (upper left) Red**
W2		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W3		**Subspans 3 and 2 of Green:** See W1 definition for pixel locations
W4		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W5		**Subspans 3 and 2 of Blue:** See W1 definition for pixel locations
W6		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W7		**Subspans 3 and 2 of Alpha:** See W1 definition for pixel locations
W8.7:1		Reserved (not written): W8 is only delivered when **Pixel Fault Mask Enable or Sample Tap Discard** is enabled.
W8.0	31:16	**Border Color Mask:** This field has the bits for all pixels set to 1 except those pixels in which a border color was source for at least one texel.
W8.0	15:0	**Pixel Null Mask:** This field has the bit for all pixels set to 1 except those pixels in which a null page was source for at least one texel.

Return Format = 16-bit

A SIMD16 writeback message with **Return Format** of 16-bit consists of up to 4 destination registers. Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in both destination registers of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0, and alpha to regid+1. The pixels written within each destination register is determined by the execution mask on the "send" instruction.

If Pixel Null Mask Enable is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bit	Description
W0.7	31:16	**Subspan 3, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Half Float, S15 signed 2's comp integer, or U16 unsigned integer.
	15:0	**Subspan 3, Pixel 2 (lower left) Red**
W0.6	31:16	**Subspan 3, Pixel 1 (upper right) Red**
	15:0	**Supspan 3, Pixel 0 (upper left) Red**
W0.5	31:16	**Subspan 2, Pixel 3 (lower right) Red**
	15:0	**Subspan 2, Pixel 2 (lower left) Red**
W0.4	31:16	**Subspan 2, Pixel 1 (upper right) Red**
	15:0	**Supspan 2, Pixel 0 (upper left) Red**
W0.3	31:16	**Subspan 1, Pixel 3 (lower right) Red**
	15:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.2	31:16	**Subspan 1, Pixel 1 (upper right) Red**
	15:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.1	31:16	**Subspan 0, Pixel 3 (lower right) Red**
	15:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.0	31:16	**Subspan 0, Pixel 1 (upper right) Red**
	15:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Green:** See W0 definition for pixel locations
W2		**Blue:** See W0 definition for pixel locations
W3		**Alpha:** See W0 definition for pixel locations
W4.7:1		Reserved (not written): W4 is only delivered when **Pixel Fault Mask Enable** is enabled.
W4.0	31:16	**Border Color Mask:** This field has the bits for all pixels set to 1 except those pixels in which a border color was source for at least one texel.
W4.0	15:0	**Pixel Null Mask:** This field has the bit for all pixels set to 1 except those pixels in which a null page was source for at least one texel.

SIMD8

Return Format = 32-bit

A SIMD8* writeback message consists of up to 4 destination registers (5 in the case of sample+killpix). Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in the destination register of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0, and alpha to regid+1. The pixels written within each destination register is determined by the execution mask on the *send* instruction.

For the sample+killpix message types, an additional register (W4) is included after the last channel register.

If **Pixel Null Mask Enable** is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bits	Description
W0.7	31:0	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Float, S31 signed 2's comp integer, or U32 unsigned integer.
W0.6	31:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.5	31:0	**Subspan 1, Pixel 1 (upper right) Red**
W0.4	31:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.3	31:0	**Subspan 0, Pixel 3 (lower right) Red**
W0.2	31:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.1	31:0	**Subspan 0, Pixel 1 (upper right) Red**
W0.0	31:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W2		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations
W3		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W4.7:1		Reserved (not written) **:** This W4 is only delivered for the sample+killpix message or when **Pixel Null Mask Enable** in Message Header is set.

DWord	Bits	Description
W4.0	31:16	**Dispatch Pixel Mask:** For his field is always 0xffff to allow dword-based ANDing with the R0 header in the pixel shader thread for sample+killpix messages.
W4.0	15:0	**Active Pixel Mask:** For sample+killpix messages this field has the bit for all pixels set to 1 except those pixels that have been killed as a result of chroma key with kill pixel mode. Since the SIMD8 message applies to only 8 pixels, only the low 8 bits within this field are used. The high 8 bits are always set to 1.
W4.0	31:24	Reserved: always written as 0xff when **Pixel Null Mask Enable** in Message Header is set.
W4.0	23:16	**Border Color Mask:** This field has the bits for all pixels set to 1 except those pixels in which a border color was source for at least one texel.
W4.0	15:8	Reserved: always written as 0xff when **Pixel Null Mask Enable** in Message Header is set.
W4.0	7:0	**Pixel Null Mask:** This field has the bit for all pixels set to 1 except those pixels in which a null page was source for at least one texel when **Pixel Null Mask Enable** in Message Header is set.

Return Format = 16-bit

A SIMD8* writeback message with **Return Format** of 16-bit consists of up to 4 destination registers). Which registers are returned is determined by the write channel mask received in the corresponding input message. Each asserted write channel mask results in the destination register of the corresponding channel being skipped in the writeback message, and all channels with higher numbered registers being dropped down to fill in the space occupied by the masked channel. For example, if only red and alpha are enabled, red is sent to regid+0, and alpha to regid+1. The pixels written within each destination register is determined by the execution mask on the send instruction.

If Pixel Null Mask Enable is enabled an additional register is included after the last channel register. Behavior of pixels that had a null texel contribution depends on the setting of **Null Pixel Behavior**.

DWord	Bits	Description
W0.7:4		Reserved
W0.3	31:16	**Subspan 1, Pixel 3 (lower right) Red:** Specifies the value of the pixel's red channel. Format = IEEE Half Float, S15 signed 2's comp integer, or U16 unsigned integer.
	15:0	**Subspan 1, Pixel 2 (lower left) Red**
W0.2	31:16	**Subspan 1, Pixel 1 (upper right) Red**
	15:0	**Supspan 1, Pixel 0 (upper left) Red**
W0.1	31:16	**Subspan 0, Pixel 3 (lower right) Red**
	15:0	**Subspan 0, Pixel 2 (lower left) Red**
W0.0	31:16	**Subspan 0, Pixel 1 (upper right) Red**
	15:0	**Supspan 0, Pixel 0 (upper left) Red**
W1		**Subspans 1 and 0 of Green:** See W0 definition for pixel locations
W2		**Subspans 1 and 0 of Blue:** See W0 definition for pixel locations

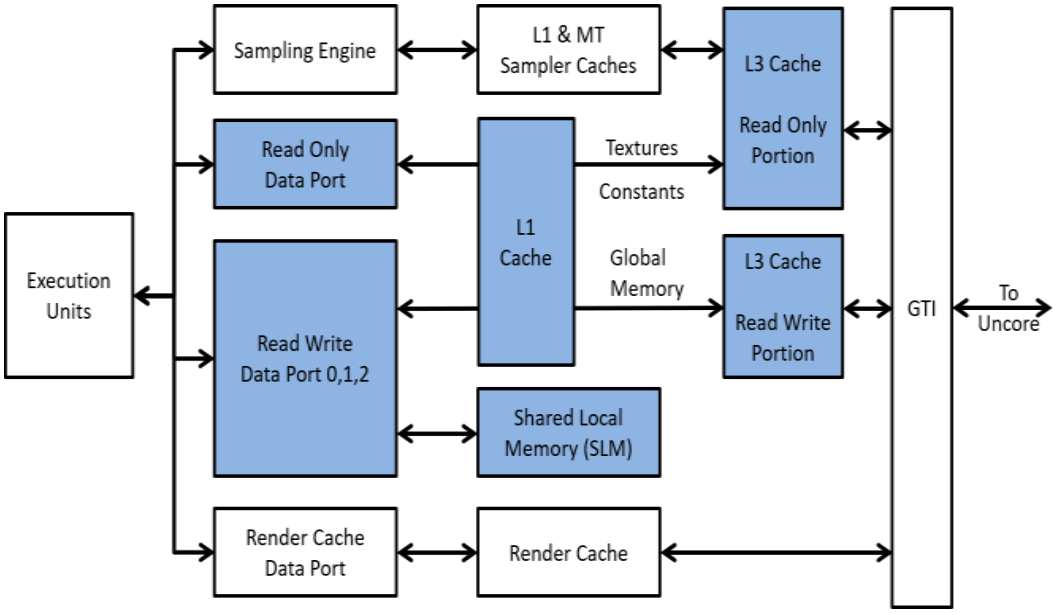
DWord	Bits	Description
W3		**Subspans 1 and 0 of Alpha:** See W0 definition for pixel locations
W4.7:1		Reserved (not written): This W4 is only delivered when **Pixel Fault Mask Enable** is enabled.
W4.0	31:24	Reserved: always written as 0xffff
W4.0	23:16	**Border Color Mask:** This field has the bits for all pixels set to 1 except those pixels in which a border color was source for at least one texel.
W4.0	15:8	Reserved: always written as 0xff
W4.0	7:0	**Pixel Null Mask:** This field has the bit for all pixels set to 1 except those pixels in which a null page was source for at least one texel.

Data Port

The Data Port provides all memory accesses for the subsystem other than those provided by the sampling engine. These include constant buffer reads, scratch space reads/writes, and media surface accesses. Render Target accesses to the Render Cache are described in the *Shared Functions Render Target* chapter.

The diagram below shows how the Read-Only and Read/Write Data Ports connect with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.

Data Port Connections to Caches and Memory



The kernel programs running in the execution units communicate with the data port via messages, the same as for the other shared function units. The data ports are considered to be separate shared functions, each with its own shared function identifier.

Read/Write Ordering

Memory access ordering in subsystem programs is managed inside of each thread. Memory access ordering between threads is managed by software because threads can be run in any order, and another thread can run when a thread is blocked waiting for data.

Ordering Conditions
Read, write, and atomic operations issued from the same thread to the same address, using the same data port, and the same cache coherency type, are guaranteed to be processed in the same order as they are issued. Software mechanisms are used to ensure ordering of accesses when issued from different threads, to different addresses, with different data ports, or with different cache coherency types. So within the same thread, memory accesses may not be ordered under some circumstances . For example: • The order that a single SIMD data port read or write message performs its multiple memory accesses is not guaranteed. • If two different virtual addresses map to the same physical memory address, the order of the operations is not guaranteed. • If a read misses a data cache and needs to fetch data from system memory, an access to a different address can complete earlier if it hits a data cache.

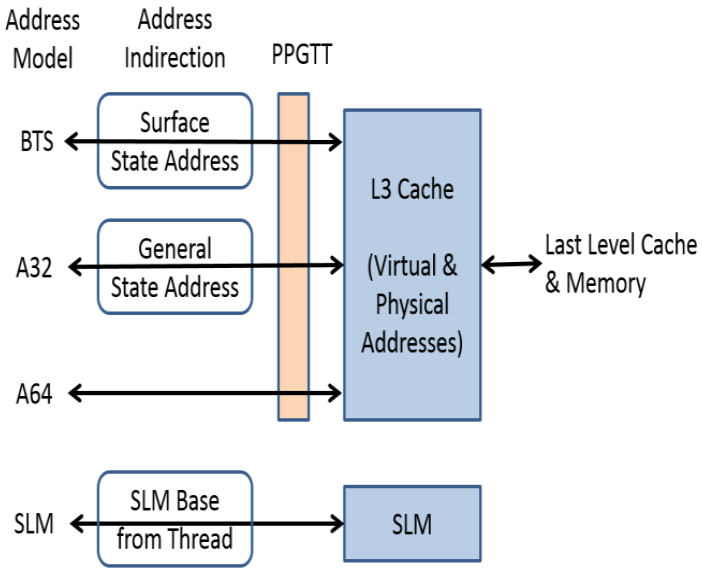
When memory access ordering is required and not guaranteed, either in the same thread or with different threads, software must ensure the ordering using fence operations or atomic operations. The can be used to block a thread until all previous messages issued to that data port cache agent have been globally observed from the point of view of other threads in the system. Different threads in the same thread group can use a Memory Fence message guarantee either global memory ordering or SLM memory ordering at the point when all the threads have reached a thread group barrier. Software can use the results of atomic operations to implement in programs a memory ordering between threads using exclusion locks or other protocols.

Address Models

Data structures accessed by the data port are called "surfaces". There are four different Address Models used by the data port to access these surfaces: Binding Table State model (BTS), Shared Local Memory model (SLM), 32-bit Stateless model (A32), and 64-bit Stateless model (A64).

Most messages support only a subset of the Address Models.

Address Models



Surface State Addresses and General State Addresses are translated to virtual Graphics Addresses using the corresponding Base Address in the STATE_BASE_ADDRESS pointers, along with additional base address offset information provided with the Data Port message. SLM addresses are offset by the SLM Base Address that is set with each thread dispatch.

The sections below explain how the Base address is calculated for each Address Model. The Data Port messages then combine that Base address with multiple SIMD offsets for the message's memory operations.

Each calculated memory address is bounds-checked against the Address Model's boundaries. Out-of-bounds read accesses return zero and out-of-bounds write accesses are dropped.

Binding Table Surface State Model (SSM and BTS)

The data port uses the binding table to bind indices to surface state, using the same mechanism used by the sampling engine.

Surface State Address Model Description
The surface state model is used when a Binding Table Index (specified in the message descriptor) of less than 240 is specified, or with the value of 252. When the Binding Table Index is less than 240, the Binding Table Index is used to index into the active binding table, and the binding table entry contains a pointer to the SURFACE_STATE. When the Binding Table Index is 252, the pointer to the SURFACE_STATE is the value of the Surface State Offset (specified in the extended message descriptor).

A Surface State Offset may be used in place of the Binding Table Index and Binding Table Pointer to form what is called a Bindless Surface State address. Throughout this volume, the shorthand terms SSM (Surface State Address Model) and BTS (Binding Table Surface Address Model) are used interchangeably.

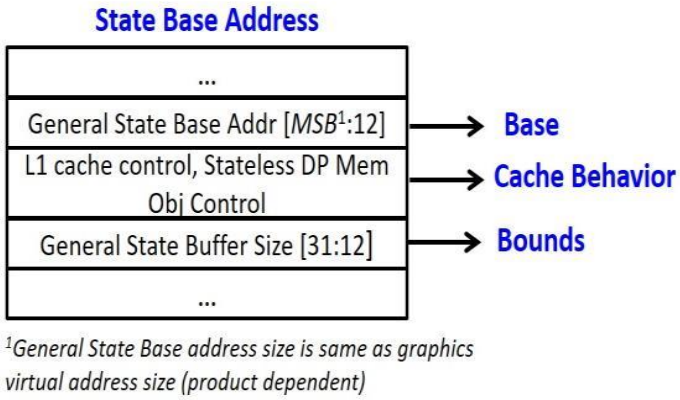
A32 Stateless Model

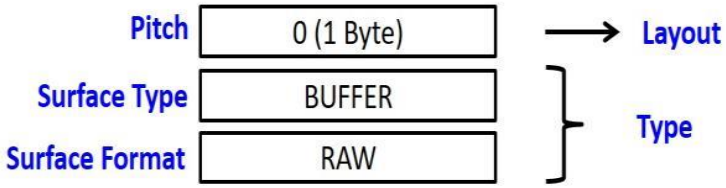
In the A32 Stateless model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = RAW
- Vertical Line Stride = 0
- Surface Base Address = General State Base Address
- Buffer Size = compared against General State Buffer Size for bounds checking
- Surface Pitch = 1 byte
- Utilize Fence = false
- Tiled = false

This model is primarily intended for scratch space buffers and for GPGPU global memory accesses.

A32 Stateless Memory Characteristics





The General State Base Address is 4KB aligned, and the bit width is product dependent, as shown below:

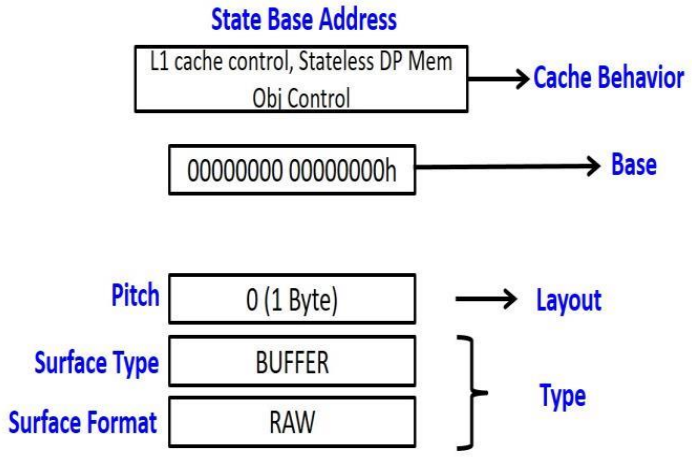
General State Base Address Bit Width
47:12

A32 Flat Model

In the A32 Flat model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = RAW
- Vertical Line Stride = 0
- Surface Base Address = 0
- Buffer Size = not checked
- Surface Pitch = 1 byte
- Utilize Fence = false
- Tiled = false

A32 Stateless Memory Characteristics



The **base address** for "A32 Flat" model is 0x0.

Bounds check is canonical range check of the resulting 64bit address (base + offset): 0x0000_0000_0000_0000 + A32_offset. This address will always pass canonical range check since the graphics virtual address size is at least 48 bits, and the offset is only 32 bit.

A64 Flat/Stateless Model

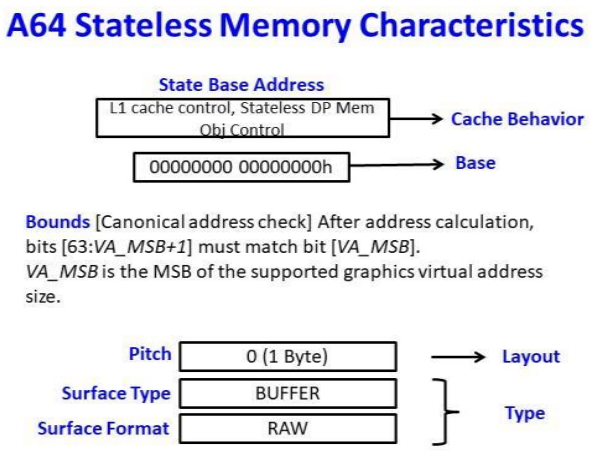
In the A64 model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = RAW
- Vertical Line Stride = 0
- Surface Base Address = Graphics Address
- Buffer Size = not checked

- Surface Pitch = 1 bytes
- Utilize Fence = false
- Tiled = false

This model is primarily intended for programmable shader programs.

A64 Stateless Memory Characteristics



Bounds check for A64 addressing is same making sure the message's 64-bit address offsets satisfies the canonical address check rule: *addr[63:VA_MSB+1]* is same as *addr[VA_MSB]*. The *VA_MSB* value depends on the supported graphics virtual address size in the product (e.g. VA_MSB==47 for 48b virtual address size). The in-bound A64 address ranges based on canonical rule is shown below:

In-bound A64 Address Ranges (canonical rule)

Low Address Range	High Address Range	Notes
0x0 - 0x7fff_ffff_ffff	0xffff_8000_0000_0000 - 0xffff_ffff_ffff_ffff	48b virtual address

If any address offset fails canonical check, all bytes accessed by that offset is treated out-of-bound.

Programming Note
If STATE_COMPUTE_MODE instruction sets Force Non-Coherent Mode, then all A32 accesses by Data Port messages are non-coherent.

Shared Local Memory (SLM)

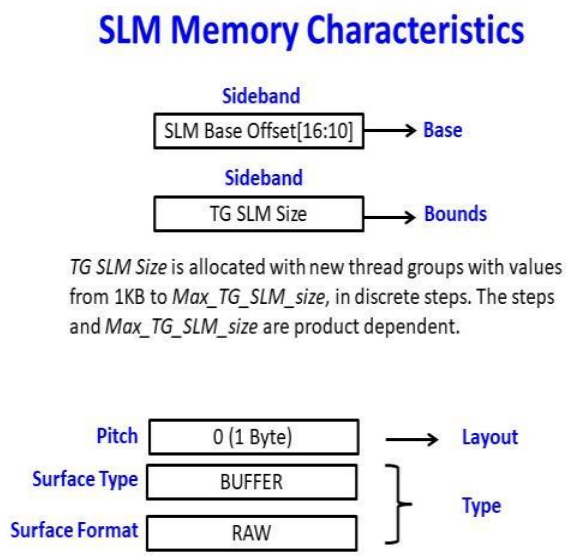
Shared local memory (SLM) is a high bandwidth memory that is not backed up by system memory. The SLM contents are shared between all active threads in the same thread group. Its contents are uninitialized after creation, and its contents disappear when deallocated.

In the SLM model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = RAW
- Surface Base Address = points to the start of the internal SLM (no memory address is applicable)
- Surface Pitch = 1 byte

Due to the predefined surface state attributes for the SLM, only a subset of the data port messages can be used.

SLM Memory Characteristics



1KB aligned "SLM Base Offset" comes on sideband and is used as the base address for SLM accesses by a message. "TG SLM Size" also comes on sideband and is used for bounds checking of SLM address(es) from the message payload. The allowed "TG SLM Size" values are as follows.

Allowed values of TG SLM Size

TG SLM Size
0KB
1KB
2KB
4KB
8KB
16KB

TG SLM Size
32KB
64KB

Specifying SLM Access

Programming Note
SLM is accessed for data ports 0 and 1 messages when a Binding Table Index (specified in the message descriptor) of 254 is specified, or by Data Port 2 messages that are not accessing A32 or A64 memory. Only the data cache data port is supported for SLM, the other data ports treat Binding Table Index 254 as a normal surface state access.
SLM is accessed when the SFID value of 0xE (data port SLM) is specified.

Programming Note
Accesses to SLM are bounds checked against the size of the SLM allocation. Addresses beyond the SLM size are out of bounds.
SLM should not be accessed through threads dispatched by the 3D pipe.

Unified Return Buffer Memory

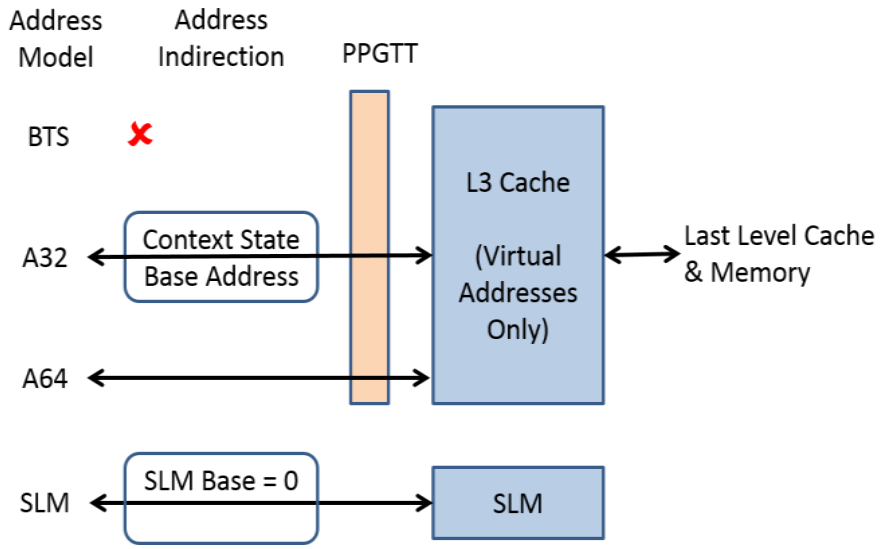
Unified Return Buffer memory (URB) is a high bandwidth memory that is not backed up by system memory. The URB contents are shared between all active threads in the same context. Its contents are uninitialized after creation, and its contents disappear when deallocated.

URB memory is read and written with specialized messages sent to URB Shared Function data port SFID_URB.

Address Models during SIP

When the SIP routine for the thread is running, only the stateless and SLM address models are supported with Data Port messages.

Address Models during SIP



The A32 base address is overridden during the SIP execution to be Dual Subslice's context save area. The SLM base is overridden during the SIP execution to be the Dual Subslice's SLM region.

Stateless A32 and A64 memory accesses are overridden to be non-coherent (L3 cache stores them as virtual addresses).

In addition to the address model restrictions during the SIP execution, only a subset of Data Port messages are defined for use during the SIP execution.

Context Save and Restore Supported Messages for SIP
All A32 stateless messages with BTI =253 override A32 base address with the Context Save area's base address during SIP. This includes the Oword Block and Oword Aligned Block messages, and the Atomic Integer Operation messages. However, Hword Scratch Read/Write messages do not override the A32 base address, and should not be used during SIP.

Overview of Memory Accesses

The specific message operation modifies the characteristics of the Address Model to determine the behaviors of each Data Port memory access. An overview of the characteristics is provided in the table below, and then explained in more detail in the following sections.

Characteristic	Overview
Surface Type	Each message has its own requirements for which surface types are supported. Most messages only operate on a restricted subset of surface types. Most Data Port messages only operate on the SURFTYPE_BUFFER surface type, but some specific messages operate on other structured surface types.

<table>
<tr><th>Characteristic</th><th>Overview</th></tr>
<tr><td>Address Alignment</td><td>Each message has its own requirements for address alignment. Most messages require an address that is aligned on a natural boundary for the width of the data item being accessed.</td></tr>
<tr><td>Address Calculation</td><td>Each message calculates one common Base Offset, and multiple additional Offsets for the SIMD operation.

Depending on the message, the SIMD Offsets can be byte address offsets, or indices that select an address aligned by the message operation's data width.</td></tr>
<tr><td>Bounds Checking</td><td>The per-message Base Address Offset is added to each of the message's additional SIMD Offsets to form offsets. Those offsets are each checked that they are within the surface's and message's boundaries before a memory access is made.

If a memory access is within bounds, each calculated offset is added to the Address model's Base Address to form a virtual, physical, or SLM address that is accessed through the L3 cache structure. If a memory access is out-of-bounds, then the message-specific action occurs (see Bounds Checking in this volume).</td></tr>
<tr><td>Canonical Address Checking</td><td>The calculated virtual memory address, formed from each calculated offset and the Address model's Base Address, could cross the canonical address boundary.

Messages using Binding Table State and A32 stateless address models require the graphics driver to correctly set up the base and bounds parameters so that every in-bounds access does not cross the canonical address boundary.

Messages using A64 Stateless address models detect any calculated offsets that cross the canonical address boundary, either as a fault or as an out-of-bounds access.</td></tr>
<tr><td>Tiled Resources Checking</td><td>The virtual memory address for a data port access may be in the range of addresses that are recognized as Tiled Resources, and that are translated through the TRTT (Tiled Resources Translation Table).

Sparse Tiled Resources support a kind of application-managed virtual memory scheme, where the application tells the driver to map specified 64KB address regions within the surface to memory resources called Tile Pools. Tiles that are not mapped to a Tile Pool are either:
Null tiles, which are treated as Out-of-Bounds
Invalid tiles, which are treated as page faults
Tiles that are mapped to a Tile Pool have a virtual address that is handled like any virtual address that is outside of the Tiled Resources address range.

<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>Cached Read of a Null Tile</td></tr><tr><td colspan="2">If a read from a Null tile gets a cache-hit in a virtually addressed GPU cache, then the read may not return zeroes.</td></tr></table></td></tr>
<tr><td>Fault Handling</td><td>Virtual memory addresses for data ports are translated through the PPGTT. If a fault occurs in</td></tr>
</table>

Characteristic	Overview
	either the fault-and-stream or fault-and-halt page fault modes, a fault handler is invoked on the IA processor to potentially correct the fault condition and retry the access. Execution of a faulted thread is suspended until the IA driver signals the GPU to restart all faulted threads and replay the faulted memory accesses. SLM accesses do not fault. See Read/Write Ordering in this volume for a description of how memory ordering is impacted by a page fault.
Out-of-Bounds handling	If a memory access is marked as out of bounds for the surface or the message, or if a virtual address maps to a Tiled Resource Null page, then a read operation returns 0 and a write operation is dropped.
Cache Behavior	Data port memory values can be cached both inside or outside the GPU. The primary GPU data cache can either be kept coherent with the IA processor with every memory access, or left non-coherent where software protocols manage the coherency. Each Address Model specifies the coherency model for the read/write data cache ports. • For Surface State accesses: the surface's RENDER_SURFACE_STATE **Coherency Type** field • For A32 and A64 Stateless accesses: the 253 and 255 aliases (see in this volume for details) • SLM accesses are localized and not cached The behavior of writing cache data to memory (e.g. write back or write through modes) is specified by the MEMORY_OBJECT_CONTROL_STATE. • For Surface State accesses: the surface's RENDER_SURFACE_STATE **Surface Object Control State** • For A32 and A64 Stateless accesses: the STATE_BASE_ADDRESS **Stateless Data Port Access Memory Object Control** When PPGTT is enabled, the page table entry overrides the LLC/eDRAM cache controls supplied by the MEMORY_OBJECT_CONTROL_STATE.

Surfaces Types

Each message has its own requirements for which surface types are supported. Most messages only operate on a limited subset of surface types. The BUFFER surface type is supported by all the address models. All other surface types are only supported by the BTS address model.

The stateless and SLM address models have a pre-defined surface type, format, and layout. Untyped messages are used with these address models.

The BTS address model is used with either typed or untyped messages. The BTS surface type must be compatible with the expected surface type, format, and layout of the message operation. Untyped messages override the surface format with their specific data type and size. Typed messages require the surface characteristics expected by the message, or the message results are undefined (accesses may be dropped or may return random data).

Surface Types used by Data Port Messages

Message Category	BTS	SLM	A32	A64
Render Target	1D, 2D, 3D, CUBE, BUFFER, NULL	N/A	N/A	N/A
Media	2D, NULL	N/A	N/A	N/A
Typed Surface or Atomic	1D, 2D, 3D, CUBE, BUFFER, NULL	N/A	N/A	N/A
Untyped Surface or Atomic	BUFFER, SCRATCH, NULL	BUFFER	BUFFER	BUFFER
Untyped Scattered or Block	BUFFER, NULL	BUFFER	BUFFER	BUFFER
Scaled Untyped Scattered or Surface	N/A	BUFFER	BUFFER	BUFFER

When a surface state is present, untyped messages ignore the surface format and treat the data as RAW. For example, data is returned from the constant buffer to the GRF without format conversion.

2D, 3D, CUBE and NULL surfaces support YMAJOR tile modes to improve memory locality of nearby accesses. All surface types support LINEAR tile mode. Surfaces with type BUFFER and SCRATCH cannot be tiled. 2D surfaces support XMAJOR tile mode for the Display engine. Stencil format is supported in WMAJOR tile mode for 2D, 3D and CUBE tile formats.

Tile Modes supported by Data Port Surface Types

Surface Type	Linear	Y Major	W Major	X Major
BUFFER	Yes			
SCRATCH	Yes			
1D	Yes			
2D	Yes	Yes	Yes	Yes
CUBE	Yes	Yes	Yes	
3D	Yes	Yes	Yes	
NULL	Yes	Yes	Yes	Yes

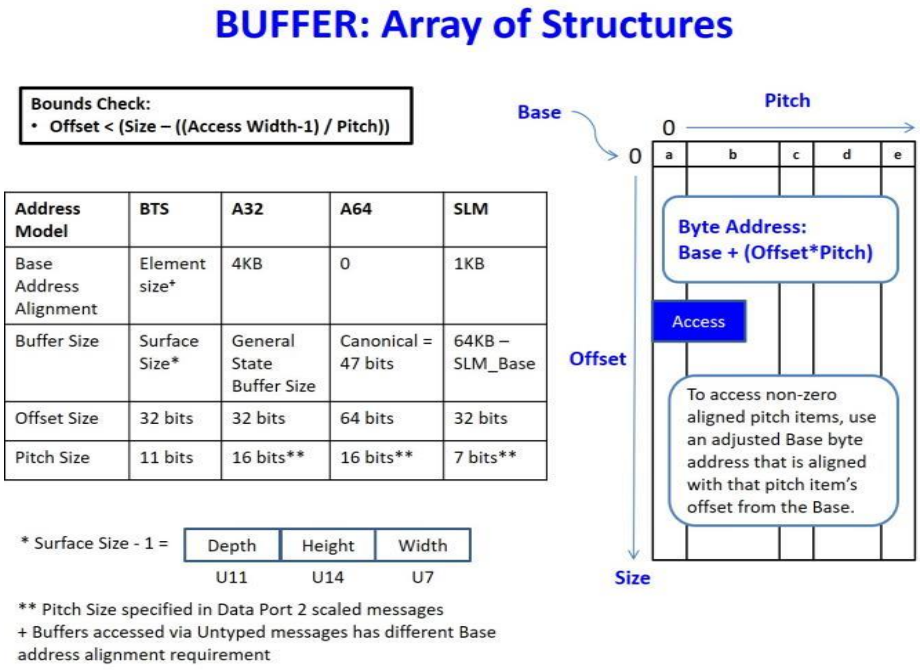
See **Surface Layout and Tiling** in the Memory Views volume for more information about tile layouts.

Programming Note	
Context:	Surfaces Types
For data port messages, compressed surface formats for MSAA and Media surfaces must be Y Major tiled (not Linear, X Major, or W major).	

Addressing Buffers (SURFTYPE_BUFFER)

Most data port messages operate on BUFFER surface types. The figure below illustrates how the base address and bounds calculations are performed on BUFFER surface types.

BUFFER: Array of Structures



Address Model	BTS	A32	A64	SLM
Base Address Alignment	Element size+	4KB	0	1KB
Buffer Size	Surface Size*	General State Buffer Size	Canonical = 47 bits	64KB – SLM_Base
Offset Size	32 bits	32 bits	64 bits	32 bits
Pitch Size	11 bits	16 bits**	16 bits**	7 bits**

Untyped Message U Offset Alignment

U address (byte offset) alignment of Untyped messages accessing SURFTYPE_BUFFER are described in the **message summary page.**

Surface Base Address Alignment for Buffers

For Buffers accessed by Untyped data-port messages, the Base Address must be aligned to MAX(4B, Msg_offset_alignment). I.e., the Surface Base Address has the same alignment requirement as the message's U offsets, but it should also be at least DW aligned.

Surface Size Alignment for Buffers

For Buffers accessed by Untyped data-port messages, the Size must be multiple of MAX(4B, Msg_offset_alignment). I.e. the Surface Base Address has the same alignment requirement as the message's U offsets, but it should also be at least DW aligned.

Accessing Typed Buffers via Untyped Messages

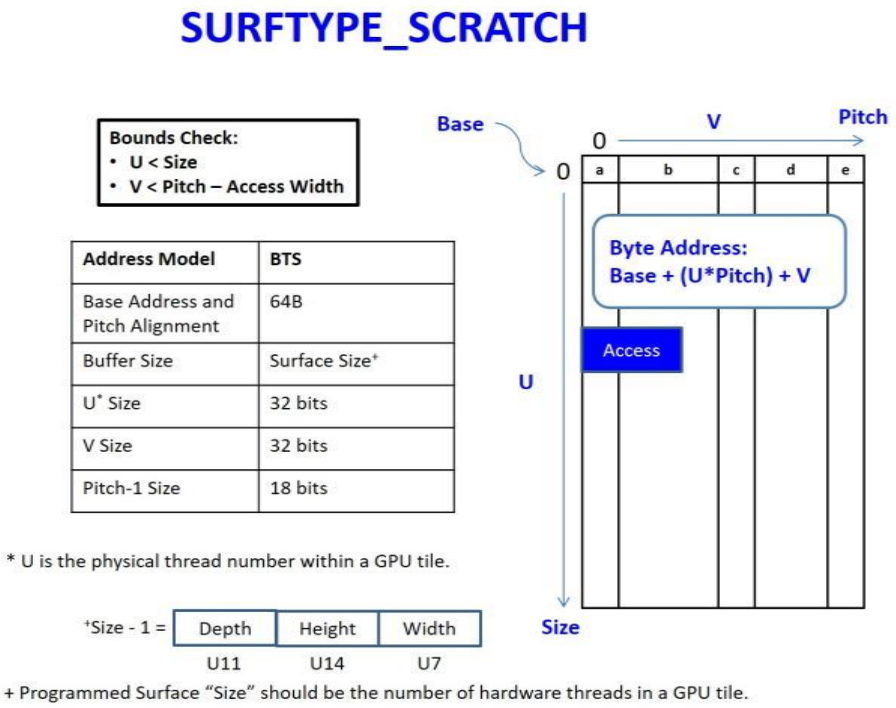
Typed Buffers (i.e. SURFTYPE_BUFFER where Pixel format is not Raw) must not be accessed via Untyped Messages.

Addressing SCRATCH Buffers (SURFTYPE_SCRATCH)

Some data port messages operate on SCRATCH surface types. The figure below illustrates how the base address and bounds calculations are performed on SCRATCH surface types.Scratch Buffers (SURFTYPE_SCRATCH) are laid out like an Array of Structures.

Programming Note
The U index to the Array of Structure is same as the unique hardware physical thread number within a GPU tile, and the V offset is supplied in the Data Port message's address payload. The GPU tile physical thread number is formed by concatenating the fields {Slice_ID, DSS_ID, ROW_ID, EU_ID, EU_THREAD_ID}. The width of these fields are dependent on the product configuration, and are expanded to the containing power-of-2 boundary. For example, if there are 12 physical threads per EU, then the field value is expanded to 16. For each field, software allocates enough entries to support the maximum physical value of the field. For example, even when a subslice is disabled in a configuration, the number of buffer entries is allocated for the total number of subslices in the tile, including the disabled ones.

SURFTYPE_SCRATCH: Array of Structures



Programming Note	
Context:	Addressing SCRATCH (SURFTYPE_SCRATCH)
Read/Write Data Ports only support linear tiled scratch surface.	

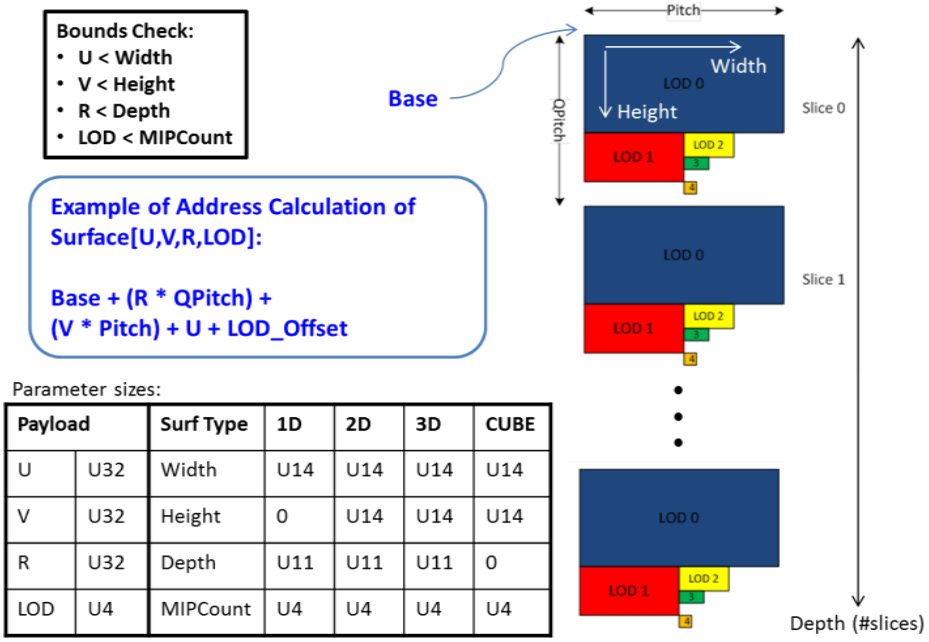
For messages accessing SURFTYPE_SCRATCH surface types, the V address must be aligned to the data access width (for example, DWords must have the low 2 bits as zeros).

Addressing 1D, 2D, 3D, CUBE Surfaces

Some data port messages operate on various BTS surface types. The addressing and bounds checking for the 1D, 2D, 3D, and CUBE surface types is fully explained in other chapters. See SURFACE_STATE in the Shared Functions volume and **Surface Layout and Tiling** in the GPU Overview volume.

The figure below illustrates the differences in addressing and bound checking between these surface types and the BUFFER and STRBUF surface types.

Multi-Dimensional Surface Buffers



Parameter sizes:

Payload		Surf Type	1D	2D	3D	CUBE
U	U32	Width	U14	U14	U14	U14
V	U32	Height	0	U14	U14	U14
R	U32	Depth	U11	U11	U11	0
LOD	U4	MIPCount	U4	U4	U4	U4

The following table indicates the hardware interpretation of each input address parameter based on surface type. If LOD is specified, then both V and R parameters must also be included, even if they are ignored. If LOD is not specified, then unused V and R parameters are optional in the address payload. If LOD is not present in the address payload, then it has the default value of 0. If the surface state indicates the Number of Multisamples > 1, then the sample number is the first parameter in the address payload.

Address Parameters Used by Surface Type

SURFACE_STATE Fields			Address Payload				
Surface Type	Surface Array	Number of Multisamples	MSAA	U Address	V Address	R Address	LOD
SURFTYPE_1D	disabled	Must be MULTISAMPLECOUNT_1	Never Present	X pixel address	(ignored if present)	(ignored if present)	Optional LOD
	enabled			X pixel address	array index	(ignored if present)	Optional LOD
SURFTYPE_2D	disabled	MULTISAMPLECOUNT_1	Never Present	X pixel address	Y pixel address	(ignored if present)	Optional LOD
	enabled			X pixel address	Y pixel address	array index	Optional LOD
	disabled	Not MULTISAMPLECOUNT_1	Sample Number	X pixel address	Y pixel address	(ignored if present)	Optional LOD
	enabled			X pixel address	Y pixel address	array index	Optional LOD

SURFACE_STATE Fields			Address Payload				
Surface Type	Surface Array	Number of Multisamples	MSAA	U Address	V Address	R Address	LOD
SURFTYPE_CUBE	disabled	Must be MULTISAMPLECOUNT_1	Never Present	X pixel address	Y pixel address	array index	Optional LOD
	enabled			X pixel address	Y pixel address	array index	Optional LOD
SURFTYPE_3D	disabled	Must be MULTISAMPLECOUNT_1	Never Present	X pixel address	Y pixel address	Z pixel address	Optional LOD
SURFTYPE_BUFFER	disabled	Must be MULTISAMPLECOUNT_1	Never Present	buffer index			
SURFTYPE_STRBUF	disabled			buffer index	byte offset		

Programming Note	
Context:	Addressing 1D, 2D, 3D, CUBE Surfaces - Workaround
If the surface state indicates the Number of Multisamples > 1, then the LOD parameter is not optional: the R and LOD parameters must be specified along with the MSAA sample number parameter.	

Programming Note	
Context:	Addressing 1D, 2D, 3D, CUBE Surfaces

- Vertical stride & Vertical Offset fields of the surface state object are only supported for 2D non-array surfaces.
- Tile W surfaces must be of format R8_UINT and only support SIMD8.

Surface Formats

Untyped messages allow direct read and write access to untyped surfaces. Untyped surfaces are always type BUFFER, and are assumed to be surface format RAW.

Typed messages allow direct read and write accesses to typed surfaces. Typed surfaces are of type SURFTYPE_1D, 2D, 3D, CUBE, or BUFFER and have a supported surface format other than RAW.

Read and write messages for typed surfaces convert the message's data type to/from the selected surface's format. The conversion rules are listed in a table below. Some surface formats are supported for typed surface writes, but are indirectly supported for typed surface reads. Indirectly supported surface formats provide the RAW pixel data in UINT formats, which enables software to complete the conversion to their defined format.

The read conversion of typed surface format to a register data format is always exact. If the destination precision is less than the source precision, a write conversion from a register data format to a typed surface format may be rounded, and denormalized or clamped.

Conversion Rules for Typed Surface Dataport Messages

Register Data Type	Surface Format Type	Read Conversion (Surface to Register)	Write Conversion (Register to Surface)
F32	FLOAT	IEEE float conversion. Normalize denorms if source is narrower. Convert SNan to QNan quietly.	IEEE float conversion. Round to even and denormalize if destination is narrower. Convert SNan to QNan quietly.
F32	SNORM, UNORM	Convert fixed point to IEEE float	Convert IEEE float to fixed point. Round to even and clamp to min/max if destination is narrower.
S31	SINT	Sign extend MSBs if source is narrower	Clamp to min/max if destination is narrower.
U32	UINT	Zero extend MSBs if source is narrower	Clamp to min/max if destination is narrower.
RAW32	Any	32 bit copy	*Not supported*
RAW16	Any	16 bit copy, zero extend MSBs	*Not supported*
RAW8	Any	8 bit copy, zero extend MSBs	*Not supported*

Typed dataport messages support a subset of all surface state formats. The formats supported by typed surface reads and writes are listed in the table below. The table shows what values and formats are returned for the components when read.

Typed surface formats with UINT require the message data in U32 format. Surface formats with SINT require the message data in S32 format. All other typed surface formats require the message data in FLOAT32 format. The surface format for the typed atomic integer operations must be R32_UINT or R32_SINT.

All typed surface formats are supported for writes, as long as the Bits Per Element is a multiple of 8 (byte). This includes all the surface formats excluding SRGB and YCRV formats. The table shows the full list of supported surface formats for writes.

For typed surface writes where the Surface Format has components that are not byte-aligned, all channels must be enabled for writing.

Reads to an unsupported surface format return undefined results. Writes to an unsupported surface format have undefined results.

When an out-of-bounds location is read by typed dataport message, a 0 is returned for any component that is present in the surface format, and the default value is returned for any component that is missing in the surface format. The default value for missing components is 0 for RGB, and either 1 or 1.0 for A.

Supported Typed Surface Read Formats

Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)	Data Returned By Typed Read			
			R	G	B	A
000	R32G32B32A32_FLOAT	128	F32	F32	F32	F32
001	R32G32B32A32_SINT	128	S31	S31	S31	S31
002	R32G32B32A32_UINT	128	U32	U32	U32	U32
080	R16G16B16A16_UNORM	64	F32	F32	F32	F32
081	R16G16B16A16_SNORM	64	F32	F32	F32	F32
082	R16G16B16A16_SINT	64	S31	S31	S31	S31
083	R16G16B16A16_UINT	64	U32	U32	U32	U32
084	R16G16B16A16_FLOAT	64	F32	F32	F32	F32
085	R32G32_FLOAT	64	F32	F32	0.0	1.0
086	R32G32_SINT	64	S31	S31	0	1
087	R32G32_UINT	64	U32	U32	0	1
0C0	B8G8R8A8_UNORM	32	*RAW32*	0	0	1
0C2	R10G10B10A2_UNORM	32	*RAW32*	0	0	1
0C7	R8G8B8A8_UNORM	32	F32	F32	F32	F32
0C9	R8G8B8A8_SNORM	32	F32	F32	F32	F32
0CA	R8G8B8A8_SINT	32	S31	S31	S31	S31
0CB	R8G8B8A8_UINT	32	U32	U32	U32	U32
0CC	R16G16_UNORM	32	F32	F32	0.0	1.0
0CD	R16G16_SNORM	32	F32	F32	0.0	1.0
0CE	R16G16_SINT	32	S31	S31	0	1
0CF	R16G16_UINT	32	U32	U32	0	1
0D0	R16G16_FLOAT	32	F32	F32	0.0	1.0
0D1	B10G10R10A2_UNORM	32	*RAW32*	0	0	1
0D3	R11G11B10_FLOAT	32	*RAW32*	0	0	1
0D6	R32_SINT	32	S31	0	0	1
0D7	R32_UINT	32	U32	0	0	1
0D8	R32_FLOAT	32	F32	0.0	0.0	1.0
100	B5G6R5_UNORM	16	*RAW16*	0	0	1
102	B5G5R5A1_UNORM	16	*RAW16*	0	0	1
104	B4G4R4A4_UNORM	16	*RAW16*	0	0	1
106	R8G8_UNORM	16	F32	F32	0.0	1.0
107	R8G8_SNORM	16	F32	F32	0.0	1.0
108	R8G8_SINT	16	S31	S31	0	1

Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)	Data Returned By Typed Read			
109	R8G8_UINT	16	U32	U32	0	1
10A	R16_UNORM	16	F32	0.0	0.0	1.0
10B	R16_SNORM	16	F32	0.0	0.0	1.0
10C	R16_SINT	16	S31	0	0	1
10D	R16_UINT	16	U32	0	0	1
10E	R16_FLOAT	16	F32	0.0	0.0	1.0
11A	B5G5R5X1_UNORM	16	*RAW16*	0	0	1
140	R8_UNORM	8	F32	0.0	0.0	1.0
141	R8_SNORM	8	F32	0.0	0.0	1.0
142	R8_SINT	8	S31	0	0	1
143	R8_UINT	8	U32	0	0	1
144	A8_UNORM	8	0.0	0.0	0.0	F32

Supported Typed Surface Write Formats

Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
000	R32G32B32A32_FLOAT	128
001	R32G32B32A32_SINT	128
002	R32G32B32A32_UINT	128
080	R16G16B16A16_UNORM	64
081	R16G16B16A16_SNORM	64
082	R16G16B16A16_SINT	64
083	R16G16B16A16_UINT	64
084	R16G16B16A16_FLOAT	64
085	R32G32_FLOAT	64
086	R32G32_SINT	64
087	R32G32_UINT	64
0C0	B8G8R8A8_UNORM	32
0C2	R10G10B10A2_UNORM	32
0C4	R10G10B10A2_UINT	32
0C7	R8G8B8A8_UNORM	32
0C9	R8G8B8A8_SNORM	32
0CA	R8G8B8A8_SINT	32
0CB	R8G8B8A8_UINT	32
0CC	R16G16_UNORM	32

Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
0CD	R16G16_SNORM	32
0CE	R16G16_SINT	32
0CF	R16G16_UINT	32
0D0	R16G16_FLOAT	32
0D1	B10G10R10A2_UNORM	32
0D3	R11G11B10_FLOAT	32
0D6	R32_SINT	32
0D7	R32_UINT	32
0D8	R32_FLOAT	32
100	B5G6R5_UNORM	16
102	B5G5R5A1_UNORM	16
104	B4G4R4A4_UNORM	16
106	R8G8_UNORM	16
107	R8G8_SNORM	16
108	R8G8_SINT	16
109	R8G8_UINT	16
10A	R16_UNORM	16
10B	R16_SNORM	16
10C	R16_SINT	16
10D	R16_UINT	16
10E	R16_FLOAT	16
11A	B5G5R5X1_UNORM	16
140	R8_UNORM	8
141	R8_SNORM	8
142	R8_SINT	8
143	R8_UINT	8
144	A8_UNORM	8

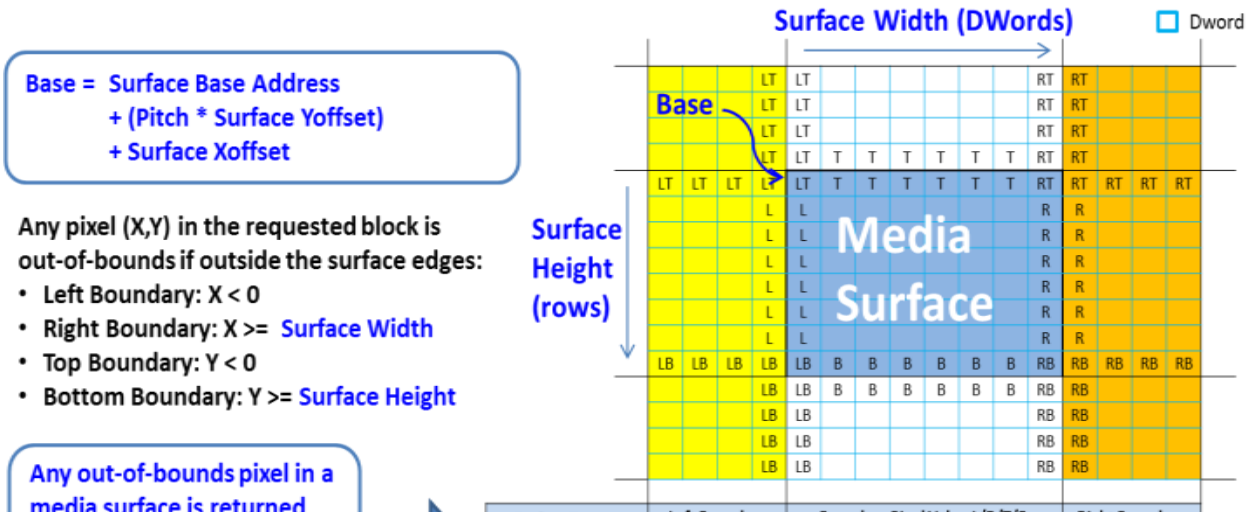
Programming Note	
Context:	Surface Formats
Indirectly supported surface formats are treated as R8_UINT for the 8 BPE formats, R16_UINT for 16 BPE formats, R32_UINT for 32 BPE formats, and R32G32_UINT for 64 BPE formats. And the Channel Mask for surface write messages, and the Shader Channel Select fields from the surface state, are interpreted with these same substitute formats.	

Addressing 2D Media Surfaces

Some data port messages operate on rectangular blocks of 2D surfaces using the BTS address model. These messages are typically used in media image processing. The rectangular block is specified in media messages as a signed (X, Y) offset from the surface's origin, and by the block's width and height.

The rectangular block specified in a media message could be partially (or fully) out-of-bounds of the surface. Some media data port messages perform pixel replication to supply values for out-of-bounds pixels. The figure below illustrates how the boundary pixel values are calculated from an in-bounds pixel value.

Media Surface Boundary Pixel Handling



Surface Format	Left Boundary	Boundary Pixel Value L/R/T/B	Right Boundary
Any 8bpe	B0B0B0B0	B0B1B2B3	B3B3B3B3
YCRCB_NORMAL	B0B1B0B3	B0B1B2B3 (Y0U0Y1V0)	B2B1B2B3
YCRCB_SWAPY	B0B1B2B1	B0B1B2B3 (U0Y0V0Y1)	B0B3B2B3
YCRCB_SWAPUV	B0B1B0B3	B0B1B2B3 (Y0V0Y1U0)	B2B1B2B3
YCRCB_SWAPUVY	B0B1B2B1	B0B1B2B3 (V0Y0U0Y1)	B0B3B2B3
Other 16bpe	B0B1B0B1	B0B1B2B3	B2B3B2B3
Any 32bpe	B0B1B2B3	B0B1B2B3	B0B1B2B3
All Others	*Not supported*	*Not supported*	*Not Supported*

Media accesses are Dwords: all 4 surface boundaries are Dword aligned
(Base is 32B aligned, Pitch is 64B aligned, Xoffset is 4*bpe aligned, Width is Dword aligned).

Upper left corner of the requested Block is specified by the signed (X, Y) coordinate.
The block offset and block width are always aligned to size of pixel (BPE) in surface.

The addressing of 2D Media surfaces is subset of the generalized surface state addressing model. For more information, see SURFACE_STATE in the Shared Functions volume and **Surface Layout and Tiling** in the GPU Overview volume.

The Vertical Stride and Vertical Stride Offset fields of the surface state object are used for 2D media surfaces to support interlaced formats in a full frame.

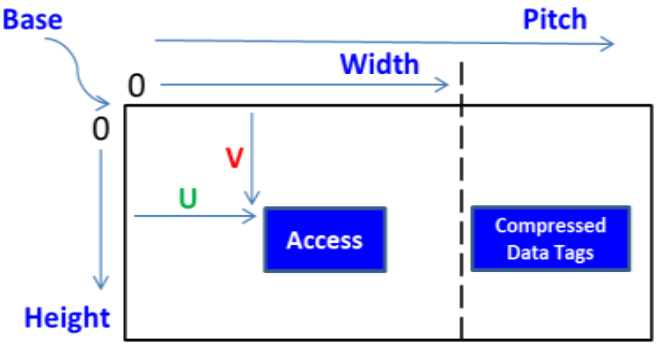
The surface's tile mode specifies how the 2D surface is laid out in the linear virtual address space. The figure below illustrates how pixel's byte address is calculated in a 2D media surface. The table lists how the surface's linear virtual address offset A[38:0] is calculated.

Media Surface Addressing

Guarantee the access is in surface bounds with Media Boundary Pixel Handling:

- **Clamp X in [0, Width-1]**
- **Clamp Y in [0, Height-1]**
- **U = Surface Xoffset + (4 * Clamp X)**
- **V = Surface Yoffset + Vertical Line Stride Offset + (Vertical Line Stride * Clamp Y)**

Byte Address: Surface_Base + Tiled_2D_Surface_Address_Offset



Media surfaces are optionally compressed, with compressed data tags in pitch region

Bits per Element	Tile Mode	Tile constants		Tiled 2D Surface Address Offset																			
		Cv	Cu	A[38:16]	A[15:12] (64KB Tile)				A[11:6] (4KB Tile)						A[5:0] (Cache Line)							
64 & 128	TileYS	6	10	u[13:10] + (v[13:6]*Pitch[17:10])	u9	v5	u8	v4	u7	v3	u6	v2	u5	u4	v1	v0	u3	u2	u1	u0		
	TileYF	4	8	u[13:8] + (v[13:4]*Pitch[17:8])																		
16 & 32	TileYS	7	9	u[13:9] + (v[13:7]*Pitch[17:9])	u8	v6	u7	v5	u6	v4	u5	v3	u4	v2								
	TileYF	5	7	u[13:7] + (v[13:5]*Pitch[17:7])																		
8	TileYS	8	8	u[13:8] + (v[13:8]*Pitch[17:8])	u7	v7	u6	v6	u5	v5	u4	v4	u3	v2								
	TileYF	6	6	u[13:6] + (v[13:6]*Pitch[17:6])																		
all	TileY	5	7	u[13:7] + (v[13:5]*Pitch[17:7])					u6	u5	u4	v4	v3	v2								
all	TileX	3	9	u[13:9] + (v[13:3]*Pitch[17:9])					v2	v1	v0	u8	u7	u6	u5	u4	u3	u2	u1	u0		
all	Linear	0	0	u[13:0] + (v[13:0]*Pitch[17:0])																		

When a 2D surface is tiled, the Surface Base Address is aligned on the tile size (4KB or 64KB).
NV12 surfaces use X/Y Offset for UV Plane surface state so that Luma 8bpe and Chroma 16bpe both aligned for TileYF/YS.

2D Media surfaces can be set up to be compressible using the Memory Compression Enable surface state field. Some media hardware will write blocks in a compressed format. The media dataport messages will read those compressed formats and automatically decompress a compressed location when read. The tag information needed to interpret and decompress data is stored in the memory region between the surface width and pitch.

Programming Note	
Context:	Addressing 2D Media Surfaces
If the surface state field Memory Compression Enable is set, then the surface state field Coherency Type must be set to GPU Coherent (not IA coherent).	

Programming Note	
Context:	Addressing 2D Media Surfaces
Writing a compressible media surface via a Data Port message will result in storing the target memory block(s) in uncompressed format.	

Programming Note	
Context:	Addressing 2D Media Surfaces

- The surface base address must be 32-byte aligned.
- The surface width must a multiple of DWords.
- Pitch must be a multiple of 64 bytes when the surface is Linear.
- Media block writes to Linear or TileX surfaces must have a height of 16 or less.
- For YUV422 formats, the block width and offset must be pixel pair aligned (i.e. DWord-aligned).
- For media block writes, both X Offset and Block Width must be DWord-aligned.
- The block width and offset must be aligned to the size of pixels stored in the surface. For a surface with 8bpe pixels for example, the block width and offset can be byte-aligned. For a surface with 16bpe pixels, it is word-aligned.

2D Media Surface Formats

Media dataport messages allow direct read and write accesses to media surfaces. Media surfaces are only type SURFTYPE_2D.

Read and write messages for media surfaces never convert the data type of the selected surface's format.

If the media surface state is set up as not compressible, the formats supported are listed in the table below. Reads or writes to an unsupported surface format have undefined results. The surface format is only used to determine out-of-bounds pixel replication.

If the media surface state is set up as compressible, then only a small subset of surface state formats are supported by the media data port message. The formats supported by compressible media surface reads are listed in the table below. Reads to an unsupported surface format return undefined results. Writes to a compressible surface format have undefined results. The surface format is used to determine out-of-bounds pixel replication and to determine compression type.

Supported Media Surface Formats

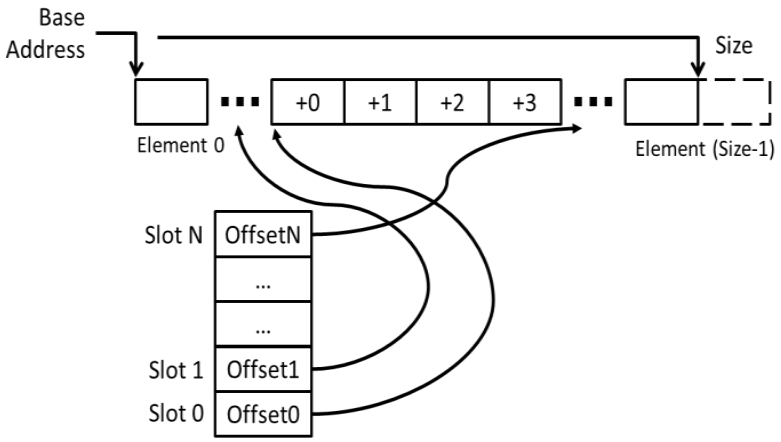
Surface Format Encoding (Hex)	Format Name	Bits Per Element(BPE)	Supported Compressible Media Surface Formats
0C0	B8G8R8A8_UNORM	32	*Not supported*
0C2	R10G10B10A2_UNORM	32	*Not supported*
0C4	R10G10B10A2_UINT	32	*Not supported*
0C7	R8G8B8A8_UNORM	32	*Not supported*
0C9	R8G8B8A8_SNORM	32	*Not supported*
0CA	R8G8B8A8_SINT	32	*Not supported*
0CB	R8G8B8A8_UINT	32	*Not supported*
0CC	R16G16_UNORM	32	*Not supported*
0CD	R16G16_SNORM	32	*Not supported*
0CE	R16G16_SINT	32	*Not supported*

Surface Format Encoding (Hex)	Format Name	Bits Per Element(BPE)	Supported Compressible Media Surface Formats
0CF	R16G16_UINT	32	*Not supported*
0D0	R16G16_FLOAT	32	*Not supported*
0D1	B10G10R10A2_UNORM	32	*Not supported*
0D3	R11G11B10_FLOAT	32	*Not supported*
0D5	R10G10B10_FLOAT_A2_UNORM	32	*Not supported*
0D6	R32_SINT	32	*Not supported*
0D7	R32_UINT	32	*Not supported*
0D8	R32_FLOAT	32	*Not supported*
100	B5G6R5_UNORM	16	*Not supported*
102	B5G5R5A1_UNORM	16	*Not supported*
104	B4G4R4A4_UNORM	16	*Not supported*
106	R8G8_UNORM	16	NV12 (UV 8bpe)
107	R8G8_SNORM	16	*Not supported*
108	R8G8_SINT	16	*Not supported*
109	R8G8_UINT	16	NV12 (UV 8bpe)
10A	R16_UNORM	16	Y16
10B	R16_SNORM	16	*Not supported*
10C	R16_SINT	16	*Not supported*
10D	R16_UINT	16	Y16
10E	R16_FLOAT	16	*Not supported*
113	A16_UNORM	16	Y16
11A	B5G5R5X1_UNORM	16	*Not supported*
140	R8_UNORM	8	NV12 (Y), Y8, IMC1 (YUV), IMC3 (YUV)
141	R8_SNORM	8	*Not supported*
142	R8_SINT	8	*Not supported*
143	R8_UINT	8	NV12 (Y), Y8, IMC1 (YUV), IMC3 (YUV)
144	A8_UNORM	8	NV12 (Y), Y8, IMC1 (YUV), IMC3 (YUV)
182	YCRCB_NORMAL	16	YUYV
183	YCRCB_SWAPUVY	16	VYUY
18F	YCRCB_SWAPUV	16	YVYU
190	YCRCB_SWAPY	16	UYVY

Slots and Elements

Data Port messages are SIMD operations: a single instruction operating over multiple data operands. Data operands are addressed either as: consecutive offsets ("elements"), with arbitrary offsets ("slots"), or in a combination of both slots and elements.

Slots and Elements



Each data port message supports a restricted subset of all possible combinations of slots and elements. The details of which combinations are supported are listed for each message in the Messages section of this volume.

The Offset in slots can be in any order and may be duplicates. The data port hardware attempts to minimize memory accesses and optimize for cache line efficiency. The hardware issues in parallel the data operations in parallel for each slot and data element pair, up to the limits of the hardware implementation. Bounds checking is performed independently on each access.

Programming Note	
Context:	Slots and Elements
Writes to overlapping addresses have undefined write ordering.	

For A32 messages, a base address offset is provided in the message header and added to each offset. The base address offset in the message header is always ignored for BTS, SLM, and A64 messages.

Typed messages always use the BTS address model and can use multiple address components to calculate the address offset for each slot. The Surface Type determines which address components are present and how the address offset is calculated.

Base Address Offsets in SIMD

Slots	SIMD Name	Location of Slot Address Offset	Description
1	Block	Message Header	Message header provides 1 address offset in that is used as the address offset of a block of data elements. Untyped blocks use 1 address. Typed blocks such as Media Block use X/Y coordinates.
2	Dual Block or SIMD4x2	Address Payload	Message address payload provides 2 address offsets which are used as the base address offset of 2 sets of data elements.
8	SIMD8	Address Payload	Message address payload provides 8 address offsets which are used as the address offset of 8 sets of data.
16	SIMD16	Address Payload	Message address payload provides 16 address offsets which are used as the address offset of 16 sets of data. Typed messages process either the upper or lower 8 slots at a time, so two messages must be used for SIMD16 operations.

Programming Note	
Context:	Slots and Elements
The current implementation limits the maximum data payload for a message to 256 bytes. Depending on the number of slots and the data width specified by the operation, some messages restrict the number of elements supported in a single SIMD operation.	

Execution Masks

Every data port message is sent with a 16-bit execution mask. The Execution Mask is used by the data port to enable the read or write operation on the mask's corresponding SIMD channel. The channel is considered active if the mask bit is set.

The execution mask is used to ensure that data associated with an inactive channel are not overwritten in the GRF (read operations) or memory (write operations).

If all execution masks are clear on the SEND instruction, the send message instruction becomes a noop and the message is not sent to the data port.

Data port messages have different semantics on how the execution mask controls the read or write accesses. The table below summarizes the semantics for all the data ports. The details of which semantics are used by each individual message is described in the Messages section of this volume.

Description of SIMD Execution Mask Semantics

Execution Mask Name	SIMD Mode and Data Type	How Execution Mask is Used
SM	SIMD8 or SIMD16	For SIMD16, the 16 bits of the execution mask control the 16 SIMD channels. For SIMD8, the lower 8 bits of the execution mask control the 8 SIMD channels, and the upper 8 bits of the execution mask are ignored.
SMBLK	SIMD8 or SIMD16 With Data Elements	For SIMD16, the 16 bits of the execution mask control the 16 SIMD channels. For SIMD8, the lower 8 bits of the execution mask control the 8 SIMD channels, and the upper 8 bits of the execution mask are ignored. If the date elements > 1, then the execution mask bits are reused for the additional data elements.
SG	SIMD8 Slot Group	Either the lower 8 bits or the upper 8 bits of the execution mask are used to control the 8 SIMD channels, based on the slot group specified in the message descriptor.
BLK	Block	The lower 8 bits of the execution mask control the first 8 DWords, and the upper 8 bits control the second 8 DWords. The DWords are accessed if the corresponding DWord's execution mask bit is set. If the block is 1 OWord size, then either the lower or upper nibbles of the lower 8 bits control the corresponding DWords in that lower or upper OWord. If the block is 8 OWords or 16 OWords, then the execution mask bits are reused for the additional channels.
BLKCM	Block with Channel Mode	The lower 8 bits of the execution mask control the first 8 DWords, and the upper 8 bits control the second 8 DWords. If the block is 1 OWord size, then either the lower or upper nibbles of the lower 8 bits control the corresponding DWords in that lower or upper OWord. If the block is 8 OWords or 16 OWords, then the execution mask bits are reused for the additional blocks.
Ignored	Not Used	The execution mask bits are ignored by this message operation. The data is controlled by other parameters to this message. The execution mask bits must be non-zero for the message to be sent to the data port.

Address Alignment and Data Widths

Data Port messages calculate multiple data operand addresses using offsets from the message's common base address. The details of how each message calculates its address is described in the Messages section.

The address calculations are described using the following notational convention:

(Base + BaseOffset) -> DataType[Offset]

This notation refers to the byte-aligned address (**Base + BaseOffset**), which is used as the address of a linear array of **DataType** operands. The **Offset** is the index into the **DataType** array with that data width.

Address Calculation

Notation	Description
Base	Each Address Model has its own base address source, illustrated in the Surfaces Types section of this volume. The Base address is architecturally defined to always be aligned at a valid address boundary for the message. The notational convention in this chapter labels this calculated base address the **Base**.
Buffer, BaseOffset	The offsets from some messages are specified as byte offsets, and others are DWord (or other DataType) offsets. **Buffer** is shorthand for Buffer Base Address offset that is specified in A32 message headers. The notation in this chapter shows byte offsets as being added to the **Base** on the left side of the pointer (->), and the DataType-sized offsets as being indices to the array on the right of the pointer. Each message defines the address alignment it operates on. Base address offsets are forced to be aligned for the message's DataType by treating any unaligned low address bits as zero. For most messages (except media block messages), the Buffer or BaseOffset value is always an unsigned (positive) value.
B[ ], W[ ], DW[ ], QW[ ], OW[ ], HW[ ]	The **DataType** array is a linear array of Bytes (**B**), Words (**W**), DWords (**DW**), QWords (**QW**), OWords (**OW**), or HWords (**HW**). The index of the array is multiplied by the width of the data type, to form a byte offset to be added to the base address. Each message defines the data type it operates on. The array's byte offset is architecturally defined to always be aligned at a valid address boundary for the message.

Notation	Description
Offset{Slot}, U{Slot}, V{Slot}	**Offset{Slot}** is shorthand for Offset0, Offset1, ... in a SIMD8 or SIMD16 address payload. **U{Slot}** and **V{Slot}** are shorthand for saying U.Offset{Slot} and V.Offset{Slot}, where U and V are SIMD8 or SIMD16 address payloads. For most messages (except media block messages), the **Offset** is always an unsigned (positive) value.
Elem, Chan	**Elem** represents the index into an array of consecutive data operands. **Chan** represents the 32-bit Red, Green, Blue, or Alpha channels of color item. These values are sequentially stored in memory as a 4 item DW block.
(Surface[U, V, R, LOD])	The notation **Surface[U, V, R, LOD]** refers to a typed surface which, depending on the surface type, may use the U, V, R, and LOD parameters to calculate the offsets from the surface's state base address for the operands.

Bounds Checking and Faulting

Bounds checking is a programming safety feature of the architecture. If a data port access is outside the boundaries of the surface, that memory operation is dropped before accessing the L3 cache. Out-of-bounds read operations return zero for the data. Each SIMD memory access is individually checked in a message operation, so some accesses could be in-bounds and while others are out-of-bounds.

Out-of-bounds checking is always performed at a DWord granularity. If any part of the DWord is out-of-bounds then the whole DWord is considered out-of-bounds.

The bounds check is performed on the Address Offset portion of the message operation. It is designed to catch user mode programming mistakes. The bounds check assumes that the surface's base address is properly aligned and that the surface's boundary limits are properly configured by the trusted kernel mode graphics driver.

The bounds check does not replace the data security features of the page tables (PPGTT). The page tables are used to protect other memory areas from accesses outside of this program's memory space. Virtual memory addresses for data ports are translated through the PPGTT, and follow that translation table's faulting model. In all page fault modes, either the fault condition is corrected, and the memory access is completed, or the entire program is terminated.

Tiled Resources
If a data port access is made to a TRTT Null tile, then that access behaves like an out-of-bounds access.

The bounds checks performed are specific to the address model and surface type of the message. See the figures in the Surfaces Types section of this volume for more information about the boundary conditions for the surfaces.

Description of Boundary Check Semantics

Bounds Check	Address Model	Surface Type	Description
Surface	BTS	BUFFER	Messages using the BTS address model detect any address offsets that, either directly or through the offset calculations, exceed the offset size of the buffer as specified in the SURFACE_STATE. When bounds checks are performed on a BTS BUFFER, Untyped Messages use the data size of the Surface element as the pitch, whereas Typed messages use the programmed surface pitch.
Surface	BTS	SCRATCH	Messages using the BTS address model detect any U address component that exceeds the U size of the structured buffer, or a V address component that exceeds the Pitch of the structured buffer, as specified in the SURFACE_STATE.
Surface	BTS	1D, 2D, 3D, CUBE	Messages using the BTS address model detect any U, V, or R address components that exceed the Width, Height, and Depth sizes of the surface, as specified in the SURFACE_STATE.
Media	BTS	2D	Some media-oriented messages using the BTS address model detect X or Y address components that exceed the Width and Height of the surface, as specified in the SURFACE_STATE. If a read access is outside of the surface boundaries, the read access is treated as an in-bounds access and returns the value of the nearest pixel. Write accesses outside of the surface boundaries are dropped.
GenState	A32	BUFFER	Messages using the A32 Stateless address model detect any address offsets that, either directly or through the offset calculations, exceed the General State Buffer Size specified in the STATE_BASE_ADDRESS. If General State Buffer Size is zero, then any A32 Stateless access is out-of-bounds.
PTSS	A32	BUFFER	Same as GenState.
Canonical	A64	BUFFER	Messages using A64 Stateless address model detect any A64 address offset in the message that cross the canonical address boundary as out-of-bound. Messages using non-A64 address models do not need to check for crossing the canonical address boundary because their surface and boundaries are assumed to be properly configured by the graphics driver.
Shared	SLM	BUFFER	Messages using the SLM address model detect any address offsets that, either directly or through the offset calculations, exceed the SLM memory size allocated to that workgroup. If any address DW(s) exceeds the allocated SLM memory size, zeros are returned for those DW(s) for Read messages, and for Write Messages those DW(s) are not updated. Only the lower 17b of SLM offset (mapped to maximum SLM space of 128KB) is considered for out-of-bound check.

Message Formats

Message operations on data ports are described using five standard parts.

Descriptor	Describes the Message Type, the Message Specific Controls, and whether a Message Header is present. The Message Descriptor and the destination Data Port (SFID) is encoded as part of the SEND instruction.
Header	When present, provides additional Message specific controls. Some messages disallow (forbid) a Header, some require a Header, and some permit an optional Header to specify additional non-default-valued parameters.
Address Payload	Provides the slot address offsets for those messages that support scattered operations.
Source Payload	Provides source data for write operations and atomic operations.
Writeback Payload	Returns result data for read operations and for atomic operations.

Most messages do not use (and therefore do not send or receive) all five parts of a message. The specific message encoded in the Message Descriptor determines which of the other four parts of the message sequence are present.

Programming Note	
Context:	Split Send with Source Payload
When a source data payload is used in dataport message, that payload must be specified as Source 1 portion of a Split Send message.	

The next sections describe all of the supported formats for the Source and Writeback Data Payloads, the Address Payloads, the Message Descriptors, and their Message Headers.

End of Thread Usage

Read/Write data port messages may not have the End of Thread bit set in the message descriptor other than the following exceptions:

The URB Write message may have End of Thread set for threads dispatched by the 3D Geometry pipe fixed functions.

Message Description Conventions

Message operations with common semantics and limitations are grouped together and described in the same section. These common characteristics are summarized in the first table in each section:

Common Characteristics of a Message Grouping

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask

Although multiple messages are described in the same section, they frequently perform their similar functions using different data ports, address models, and with different SIMD combinations of slots and

blocks. The second table in each message section lists all the valid combinations of message parameters and payloads for the messages:

Valid Message Parameters and Payloads

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload

Description of Message Summary Tables

Column Heading	Description
Addr Align	Specifies the required address alignment of the operation. This can be byte (B), DWord (DW), QWord (QW), OWord (OW), or HWord (HW). In most cases, the address must be aligned for the data width of the operation.
Data Width	Specifies the width of the data operation. This can be byte (B), DWord (DW), QWord (QW), OWord (OW), or HWord (OW).
R/W	Specifies if the operation is a Read or Write (R or W). Atomic operations that return data are classified here as reads, and ones that don't are classified here as writes.
Address Model	Specifies the address model used by this message operation: BTS, SLM, A32, or A64. Read-only data port accesses to the constant cache is labeled "CC BTS". Specifies the address model used by this message operation: BTS, SLM, A32, or A64. Read-only data port accesses to the constant cache or sampler cache are labeled "CC BTS" or "SC BTS".
Surface Type	Specifies the supported surface type. See the Surfaces Types section of this volume.
SIMD Slots	In the first table, this summarizes all the SIMD slot configurations supported for this operation and address model. In the second table, this specifies the specific SIMD slot configuration supported by the specific message operation.
Data Elements	Specifies all the sizes of data elements supported by the message operation. For blocks, this is the number of blocks. For surfaces, this is the list of channels that are enabled in the message descriptor.
SIMD Address Calculation	Describes the address calculation performed by this message operation. The calculation is usually related to the others in this section, but the details can change due to the address model used. See Address Alignment and Data Widths for an explanation of the notation used here.
Bounds Check	Specifies the bounds check performed by this message operation. The bounds check is generally determined by the address model used, but some messages change the checking. See Bounds Checking and Faulting for an explanation of the notation used here.
Execution Mask	Most message operations perform their operation on a data channel only when the corresponding execution mask is set. See Execution Masks for an explanation of the notation used here.
Message Specific Descriptor	Each data port specifies a unique encoding for its message operations and its parameters. See Message Specific Descriptors for the full list of specific message descriptors used by the data ports. In each section, all the legal combinations of messages and parameters are listed. In some cases, there are multiple message operations that perform the same function.

Column Heading	Description
Message Header	Many messages use a message header to provide additional parameters to control their operation, in addition to the parameters in the message descriptor. Each section's table lists which message headers are legal to use. See Message Headers for a full list of the message headers that are supported by all the messages.
Address Payload	Most messages perform SIMD operations using address offsets for the operation from the address payload. See Message Address Payloads for a full list of the message address payloads that are supported by all the messages. Each section's table lists which message headers are legal to use.
Source Payload	Write messages and atomic messages generally provide data in a source data payload. See Message Data Payloads for a full list of the message data payloads that are supported by all the messages. Each section's table lists which message source payloads are legal to use.
Writeback Payload	Read messages generally return data in a writeback data payload. See Message Data Payloads for a full list of the message data payloads that are supported by all the messages. Each section's table lists which message writeback payloads are used.

Messages

The message operations on data ports are described this section. The operations are organized into subsections by their addressing operation group (blocks, scattered, surfaces, atomic operations), and then by their data width and type. Each subsection describes the messages that have common semantics and limitations. (For example, read and write operations are described in the same subsection.)

Summary of Message Groups

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
Scattered and Block R/W					1, 2	1, 2, 4, 8		
Byte Scattered R/W	B	B	BTS	BUFFER, NULL	8, 16	1, 2, 4	Surface	SMBLK
	B	B	SLM	BUFFER	8, 16	1, 2, 4	Shared	SMBLK
	B	B	A32	BUFFER	8, 16	1, 2, 4	PTSS	SMBLK
	B	B	A64	BUFFER	8, 16	1, 2, 4	Canonical	SMBLK
DW Scattered R/W	DW	DW	BTS	BUFFER, NULL	8, 16	1	Surface	SM
	DW	DW	A32	BUFFER	8, 16	1	PTSS	SM
	DW	DW	A64	BUFFER	8, 16	1, 2, 4, 8	Canonical	SMBLK
QW Scattered R/W	QW	QW	A64	BUFFER	8, 16	1, 2, 4	Canonical	SMBLK
OW Block R/W	OW	OW	BTS	BUFFER, NULL	1	1, 2, 4, 8	Surface	BLK
	OW	OW	BTS	SCRATCH, NULL	1	1, 2, 4, 8	Surface	SM
	OW	OW	SLM	BUFFER	1	1, 2, 4, 8	Shared	SMBLK
	OW	OW	A32	BUFFER	1	1, 2, 4, 8	PTSS	BLK
	OW	OW	A64	BUFFER	1	1, 2, 4, 8	Canonical	BLK

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
OW Aligned Block R/W	DW	OW	BTS	BUFFER, NULL	1	1, 2, 4, 8	Surface	Ignored
	DW	OW	A32	BUFFER	1	1, 2, 4, 8	PTSS	Ignored
	DW	OW	A64	BUFFER	1	1, 2, 4, 8	Canonical	Ignored
HW Block R/W	HW	HW	BTS	SCRATCH, NULL	1	1, 2, 4, 8	Surface	SM
	HW	HW	A32	BUFFER	1	1, 2, 4, 8	PTSS	BLKCM
HW Aligned Block R/W	HW	HW	BTS	SCRATCH, NULL	1	1, 2, 4, 8	Surface	Ignored
	HW	HW	A64	BUFFER	1	1, 2, 4, 8	Canonical	BLKCM
Surface R/W					2, 8, 16	1		
Scattered Untyped Surface R/W	DW	DW	BTS	BUFFER, NULL	8, 16	{Chan 1-4}	Surface	SM
	DW	DW	BTS	SCRATCH, NULL	8, 16	{Chan 1-4}	Surface	SM
	DW	DW	SLM	BUFFER	8, 16	{Chan 1-4}	Shared	SM
	DW	DW	A32	BUFFER	8, 16	{Chan 1-4}	GenState	SM
	DW	DW	A64	BUFFER	8, 16	{Chan 1-4}	Canonical	SM
Scattered Typed Surface R/W	B	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	{Chan 1-4}	Surface	SG
Scattered MSAA Typed Surface R/W	B	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	{Chan 1-4}	Surface	SG
Atomic Operations					2, 8, 16	1		
W Untyped Atomic Integer	W	W	BTS	BUFFER, NULL	8, 16	1	Surface	SM
	W	W	SLM	BUFFER	8, 16	1	Shared	SM
	W	W	A32	BUFFER	8, 16	1	GenState	SM
	W	W	A64	BUFFER	8	1	Canonical	SM
DW Untyped Atomic Integer	DW	DW	BTS	BUFFER, NULL	8, 16	1	Surface	SM
	DW	DW	SLM	BUFFER	8, 16	1	Shared	SM
	DW	DW	A32	BUFFER	8, 16	1	GenState	SM
	DW	DW	A64	BUFFER	8	1	Canonical	SM
QW Untyped Atomic Integer	QW	QW	BTS	BUFFER, NULL	8, 16	1	Surface	SM
	QW	QW	A32	BUFFER	8, 16	1	GenState	SM
	QW	QW	A64	BUFFER	8	1	Canonical	SM
W Untyped Atomic Float	W	W	BTS	BUFFER, NULL	8, 16	1	Surface	SM
	W	W	SLM	BUFFER	8, 16	1	Shared	SM
	W	W	A32	BUFFER	8, 16	1	GenState	SM
	W	W	A64	BUFFER	8	1	Canonical	SM
DW Untyped	DW	DW	BTS	BUFFER, NULL	8, 16	1	Surface	SM

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
Atomic Float	DW	DW	SLM	BUFFER	8, 16	1	Shared	SM
	DW	DW	A32	BUFFER	8, 16	1	GenState	SM
	DW	DW	A64	BUFFER	8	1	Canonical	SM
W Typed Atomic Integer	W	W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	1	Surface	SG
DW Typed Atomic Integer	DW	DW	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	1	Surface	SG
W Atomic Counter	W	W	BTS	Any	8	1	Ignored	SG
DW Atomic Counter	DW	DW	BTS	Any	8	1	Ignored	SG
Media Block R/W								
Media Block R/W	DW	DW	BTS	2D, NULL	1	Height x Width	Media	Ignored
Others								
Read Surface Info	B	B	BTS	Any	1	1	Ignored	Ignored

Scattered and Block Read/Write Messages

This section lists the read and write messages that support scattered and block accesses of the standard untyped data: Byte, Word, DWord, QWord, OWords, and HWords.

Byte Scattered ReadWrite Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
B	B	R/W	BTS	BUFFER, NULL	8, 16	1, 2, 4	(Base + GlobalOffset + Offset{Slot})->B[Elem]	Surface	SMBLK
B	B	R/W	SLM	BUFFER	8, 16	1, 2, 4	(Base + Offset{Slot})->B[Elem]	Shared	SMBLK
B	B	R/W	A32	BUFFER	8, 16	1, 2, 4	(Base + Buffer + GlobalOffset + Offset{Slot})->B[Elem]	PTSS	SMBLK
B	B	R/W	A64	BUFFER	8, 16	1, 2, 4	(0 + Offset{Slot})->B[Elem]	Canonical	SMBLK

This message reads or writes 8 or 16 scattered and possibly misaligned Bytes, Words, or DWords, starting at each Offset. The offsets are byte-aligned. The **Global Offset** is added to each of the specific offsets when the message header is present.

A BTS surface is interpreted as a RAW BUFFER regardless of the surface format. The pitch is 1 byte and data is returned from the buffer to the GRF without format conversion.

For BTS BUFFER accesses, the Surface Base address must be aligned to DW.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Applications:

- Byte aligned buffer accesses in programmable shaders

Programming Note	
Context:	Byte Scattered Read/Write Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER, NULL	SIMD8	{1, 2, 4}	MSD0R_BS	{Forbidden}	MAP32B_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD8	{1, 2, 4}	MSD0W_BS	{Forbidden}	MAP32B_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER, NULL	SIMD16	{1, 2, 4}	MSD0R_BS	{Forbidden}	MAP32B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	BTS	BUFFER, NULL	SIMD16	{1, 2, 4}	MSD0W_BS	{Forbidden}	MAP32B_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD8	{1, 2, 4}	MSD0R_BS	{Forbidden}	MAP32B_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	{1, 2, 4}	MSD0W_BS	{Forbidden}	MAP32B_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	{1, 2, 4}	MSD0R_BS	{Forbidden}	MAP32B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	{1, 2, 4}	MSD0W_BS	{Forbidden}	MAP32B_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	{1, 2, 4}	MSD0R_BS	{Forbidden}	MAP16B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	{1, 2, 4}	MSD0W_BS	{Forbidden}	MAP16B_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	{1, 2, 4}	MSD0R_BS	{Forbidden}	MAP32B_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	{1, 2, 4}	MSD0W_BS	{Forbidden}	MAP32B_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	{1, 2, 4}	MSD0R_BS	{Forbidden}	MAP32B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{1, 2, 4}	MSD0W_BS	{Forbidden}	MAP32B_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1R_A64_BS	{Forbidden}	MAP64B_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1W_A64_BS	{Forbidden}	MAP64B_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD16	{1, 2, 4}	MSD1R_A64_BS	{Forbidden}	MAP64B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	A64	BUFFER	SIMD16	{1, 2, 4}	MSD1W_A64_BS	{Forbidden}	MAP64B_SIMD16	{1, 1, 1} MDP_DW_SIMD16	{Forbidden}

Programming Note	
Context:	Byte Scattered ReadWrite Messages
While the hardware does check for and optimize for cases where offsets are equal or contiguous, for optimal performance in some cases an aligned message with aligned data offsets may provide higher performance.	

Programming Note	
Context:	Byte Scattered ReadWrite Messages
For bounds checking the byte scattered read and write messages, the buffer size must be a multiple of 4 bytes.	

DWord Scattered Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	BTS	BUFFER,NULL	8, 16	1	Base->DW[GlobalOffset+Offset{Slot}+0]	Surface	SM
DW	DW	R/W	A32	BUFFER	8, 16	1	(Base+Buffer)->DW[GlobalOffset+Offset{Slot}+0]	PTSS	SM
DW	DW	R/W	A64	BUFFER	8, 16	1, 2, 4	(0+Offset{Slot})->DW[Elem]	Canonical	SMBLK

This message reads or writes 8 or 16 scattered DWords starting at each offset. The **Global Offset** is added to each of the specific offsets when it is provided in the message header.

A BTS surface is interpreted as a RAW BUFFER regardless of the surface format. The pitch is 1 byte and data is returned from the buffer to the GRF without format conversion.

For BTS BUFFER accesses, the Surface Base address must be aligned to DW.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Applications:

- SIMD8/16 constant buffer reads where the indices of each pixel are different (read one channel per message)
- SIMD8/16 scratch space reads/writes where the indices are different (read/write one channel per message)
- General purpose DWord scatter/gathering, used by media

Programming Note	
Context:	DWord Scattered Read/Write Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	SIMD8	1	MSD0R_DWS	{Forbidden}	MAP32B_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD0W_DWS	{Forbidden}	MAP32B_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD0R_DWS	{Forbidden}	MAP32B_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD0W_DWS	{Forbidden}	MAP32B_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD0R_DWS	{Forbidden}	MAP32B_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD0W_DWS	{Forbidden}	MAP32B_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD0R_DWS	{Forbidden}	MAP32B_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD0W_DWS	{Forbidden}	MAP32B_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1R_A64_DWS	{Forbidden}	MAP64B_SIMD8	{Forbidden}	{1, 2, 4} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1W_A64_DWS	{Forbidden}	MAP64B_SIMD8	{1, 2, 4} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD16	{1, 2, 4}	MSD1R_A64_DWS	{Forbidden}	MAP64B_SIMD16	{Forbidden}	{1, 2, 4} MDP_DW_SIMD16
W	A64	BUFFER	SIMD16	{1, 2, 4}	MSD1W_A64_DWS	{Forbidden}	MAP64B_SIMD16	{1, 2, 4} MDP_DW_SIMD16	{Forbidden}

Programming Note	
Context:	DWord Scattered Read/Write Messages
For BTS and A32 address models, the offsets in the address payload must in the range 0 - 3FFFFFFFh. If the upper 2 bits are non-zero, the result is undefined.	

QWord Scattered Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
QW	QW	R/W	A64	BUFFER	8, 16	1, 2, 4	(0+Offset{Slot})->QW[Elem]	Canonical	SMBLK

This message reads or writes 8 or 16 scattered QWords starting at each offset. The **Global Offset** is added to each of the specific offsets when it is provided in the message header.

A BTS surface is interpreted as a RAW BUFFER regardless of the surface format. The pitch is 1 byte and data is returned from the buffer to the GRF without format conversion.

For BTS BUFFER accesses, the Surface Base address must be aligned to QW.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Applications:

SIMD8/16 writes where the indices differ (write one channel per message).

General purpose QWord scatter/gathering, used by double precision compute.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1R_A64_QWS	{Forbidden}	MAP64B_SIMD8	{Forbidden}	{1, 2, 4} MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1W_A64_QWS	{Forbidden}	MAP64B_SIMD8	{1, 2, 4} MDP_QW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD16	{1, 2}	MSD1R_A64_QWS	{Forbidden}	MAP64B_SIMD16	{Forbidden}	{1, 2} MDP_QW_SIMD16
W	A64	BUFFER	SIMD16	{1, 2}	MSD1W_A64_QWS	{Forbidden}	MAP64B_SIMD16	{1, 2} MDP_QW_SIMD16	{Forbidden}

OWord Block Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
OW	OW	R/W	BTS	BUFFER,NULL	1	1, 2, 4, 8	Base->OW[GlobalOffset+Elem]	Surface	BLK
OW	OW	R/W	BTS	SCRATCH,NULL	1	1, 2, 4, 8	(Base+(TID*Pitch))->OW[GlobalOffset+Elem]	Surface	SM
OW	OW	R/W	SLM	BUFFER	1	1, 2, 4, 8	Base->OW[GlobalOffset+Elem]	Shared	BLK
OW	OW	R/W	SLM	BUFFER	1	16	Base->OW[GlobalOffset+Elem]	Shared	BLK
OW	OW	R/W	A32	BUFFER	1	1, 2, 4, 8	(Base+Buffer)->OW[GlobalOffset+Elem]	PTSS	BLK
OW	OW	R/W	A64	BUFFER	1	1, 2, 4, 8	(Base+BlockOffset0)->OW[Elem]	Canonical	BLK

This message takes one offset (Global Offset), and reads or writes 1, 2, 4, or 8 contiguous OWords starting at that offset.

A BTS surface is interpreted as a RAW BUFFER regardless of the surface format.

For BTS BUFFER accesses, the pitch is 1 byte and data is returned from the buffer to the GRF without format conversion.

For BTS BUFFER accesses, the Surface Base address must be aligned to OW.

For SCRATCH accesses, the pitch is from the surface state and the data is returned from the buffer to the GRF without format conversion.

Programming Note	
Context:	
The execution mask bits may disable accesses on the corresponding DWs of the message payload.	

Programming Note	
Context:	
For A32 and BTS OW Block messages, if (Block_offset + Block_Size) in bytes exceeds (2^32 - 1), the portion of the block that exceeds this value will have undefined return value for read messages, and data will not be written to memory for write messages.	

Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Applications:

- Constant buffer reads of a single constant or multiple contiguous constants.
- Scratch space reads/writes.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD0R_OWB	MH_BTS_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
W	BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD0W_OWB	MH_BTS_GO	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4	
R	BTS	SCRATCH,NULL	1	{1, 2, 4, 8}	MSD0R_OWB	MH_BTS_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
W	BTS	SCRATCH,NULL	1	{1, 2, 4, 8}	MSD0W_OWB	MH_BTS_GO	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4	
R	SLM	BUFFER	1	{1, 2, 4, 8}	MSD0R_OWB	MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
W	SLM	BUFFER	1	{1, 2, 4, 8}	MSD0W_OWB	MH_SLM_GO	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4	
R	SLM	BUFFER	1	{16}	MSD0R_OWB	MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
W	SLM	BUFFER	1	{16}	MSD0W_OWB	MH_SLM_GO	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4	
R	A32	BUFFER	1	{1, 2, 4, 8}	MSD0R_OWB	MH_A32_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	A32	BUFFER	1	{1, 2, 4, 8}	MSD0W_OWB	MH_A32_GO	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4	
R	A64	BUFFER	1	{1, 2, 4, 8}	MSD1R_A64_OWB	MH_A64_OWB	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
W	A64	BUFFER	1	{1, 2, 4, 8}	MSD1W_A64_OWB	MH_A64_OWB	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4	
R	CC BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD_CC_OWB	MH_BTS_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4

For the one-constant case, either the high or low half of the payload register is used depending on the half selected in Block Size, and the other half of the payload register is ignored.

Programming Note	
Context:	OWord Block Read/Write Messages
For the BTS and A32 address models, the offsets in the address payload must be in the range 0 - 0FFFFFFFh. If the upper 4 bits are non-zero, the result is undefined.	

OWord Aligned Block Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	OW	R	BTS	BUFFER, NULL	1	1, 2, 4, 8	(Base+GlobalOffset)->OW[Elem]	Surface	Ignored
DW	OW	R	A32	BUFFER	1	1, 2, 4, 8	(Base+Buffer+GlobalOffset)->OW[Elem]	PTSS	Ignored
DW	OW	R	A64	BUFFER	1	1, 2, 4, 8	(0+BlockOffset0)->OW[Elem]	Canonical	Ignored

Functional Description
This message takes one DWord-aligned offset (Global Offset), and reads 1, 2, 4, or 8 contiguous OWords starting at that offset.

A BTS surface is interpreted as a RAW BUFFER regardless of the surface format.

For BTS BUFFER accesses, the pitch is 1 byte and data is returned from the buffer to the GRF without format conversion.

For BTS BUFFER accesses, the Surface Base address must be aligned to DW.

This message is identical to the OWord Block Read/Write message except the offset alignment. The Global Offset is specified in bytes, and must be aligned to the Addr Align entry in table.

Programming Note
For A32 and BTS OW Aligned Block messages, if (Block_offset + Block_Size) in bytes exceeds (2^32 - 1), the portion of the block that exceeds this value will have undefined return value for read messages, and data will not be written to memory for write messages.

The execution mask is ignored for this message. Out-of-bounds accesses are dropped or return zero.

Applications:

Block accesses directly using a byte offset instead of an Oword index.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD0R_OWAB	MH_BTS_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
R	SLM	BUFFER	1	{1, 2, 4, 8}	MSD0R_OWAB	MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
R	A32	BUFFER	1	{1, 2, 4, 8}	MSD0R_OWAB	MH_A32_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
R	A64	BUFFER	1	{1, 2, 4, 8}	MSD1R_A64_OWAB	MH_A64_OWB	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
R	CC BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD_CC_OWAB	MH_BTS_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
R	SC BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD_SC_OWAB	MH_BTS_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4

Programming Note
For read/write cache, only the read path supports the unaligned OWord Block access.

HWord Scratch Block Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
HW	HW	R/W	BTS	SCRATCH,NULL	1	1, 2, 4, 8	(Base+(TID*Pitch))->HW[SB_Offset+Elem]	Surface	SM

This message performs a read or write operation of between 1 and 8 registers to an HWord-aligned offset. The required message header payload matches the thread payload R0.

For BTS BUFFER accesses, the Surface Base address must be aligned to HW.

Programming Note
The execution mask bits may disable accesses on the corresponding DWs of the message payload.

Out-of-bounds accesses are dropped or return zero.

Applications:

Scratch space reads/writes for register spill/fill operations.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	SCRATCH,NULL	1	{1, 2, 4, 8}	MSD0R_SB	MH_A32_HWB	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
W	BTS	SCRATCH,NULL	1	{1, 2, 4, 8}	MSD0W_SB	MH_A32_HWB	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8	{Forbidden}

Channel Mode Interpretation
Channel-enable interpretation is fixed (not programmable): • DWord, which supports a SIMD8 or SIMD16 DWord channel-serial view of a register

Programming Note
The Scratch Buffer is a bindless surface specified in the message header payload, with the base address calculated using the physical thread ID and the Pitch in the surface state. The bindless surface is relative to the Surface State Base Address heap (BTS 251). For the MSD0R_SB and MSD0W_SB messages, the HWord offset into the Scratch Buffer memory is provided in the message descriptor and communicated to the data port on the Sideband. This allows a single instruction read or write block operation in a single source instruction. The message descriptor provides 12 bits for the HWord offset, allowing addressing of 4K HWord locations (128KB).

HWord Aligned Block Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
HW	HW	R/W	BTS	BUFFER,NULL	1	1, 2, 4, 8	(Base+GlobalOffset)->HW[Elem]	Shared	Ignored
HW	HW	R/W	BTS	SCRATCH,NULL	1	1, 2, 4, 8	(Base+(TID*Pitch)+GlobalOffset)->HW[Elem]	Shared	Ignored
HW	HW	R	SLM	BUFFER	1	1, 2, 4, 8	(Base+GlobalOffset)->HW[Elem]	Shared	Ignored
DW	HW	R/W	A64	BUFFER	1	1, 2, 4, 8	(0+BlockOffset0)->HW[Elem]	Canonical	BLKCM

Functional Description
This message takes one HWord-aligned offset (Global Offset), and reads or writes 1, 2, 4, or 8 contiguous HWords starting at that offset.

A BTS surface is interpreted as a RAW BUFFER regardless of the surface format.

For BTS BUFFER accesses, the pitch is 1 byte and data is returned from the buffer to the GRF without format conversion, and the Surface Base address must aligned to DW.

For BTS SCRATCH accesses, the pitch is from the surface state and data is returned from the buffer to the GRF without format conversion.

This message is identical to the HWord Block message except the offset alignment. The Global Offset is specified in bytes, and must be aligned to the Addr Align entry in the table.

For address models A32, BTS and SLM, execution mask is ignored for this message. Out-of-bounds accesses are dropped or return zero.

Programming Note
For A32 and BTS OW Block messages, if (Block_offset + Block_Size) in bytes exceeds (2^32 - 1), the portion of the block that exceeds this value will have undefined return value for read messages, and data will not be written to memory for write messages.

Applications:

Block accesses directly using a byte offset instead of an Hword index.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	1	{1, 2, 4, 8}	MSD0R_HWAB	MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
R	BTS	SCRATCH,NULL	1	{1, 2, 4, 8}	MSD0R_HWAB	MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
R	SLM	BUFFER	1	{1, 2, 4, 8}	MSD0R_HWAB	MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
R	A64	BUFFER	1	{1, 2, 4, 8}	MSD1R_A64_HWB	MH_A64_HWB	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
W	A64	BUFFER	1	{1, 2, 4, 8}	MSD1W_A64_HWB	MH_A64_HWB	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8	

Channel Mode Interpretation
Channel-enable interpretation is fixed (not programmable): • DWord, which supports a SIMD8 or SIMD16 DWord channel-serial view of a register

Surface ReadWrite Messages

The surface read and write messages allow direct read/write accesses to untyped and typed surfaces.

Untyped surfaces are either SURFTYPE_BUFFER (format RAW, pitch 1) or SURFTYPE_SCRATCH (format RAW, pitch from the surface state). SLM and Stateless messages are always untyped BUFFER accesses.

Typed surfaces are 1D, 2D, 3D, CUBE, or BUFFER types. The surface state specifies all the parameters. There are some limitations on what typed data surface formats are supported by data ports; see the specific messages and Surfaces Types for more details.

Untyped Surface ReadWrite Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	BTS	BUFFER,NULL	8, 16	{Chan 1-4}	(Base+U{Slot})->DW[Chan]	Surface	SM
DW	DW	R/W	BTS	SCRATCH,NULL	8, 16	{Chan 1-4}	(Base+(TID*Pitch)+U{Slot})->DW[Chan]	Surface	SM
DW	DW	R/W	SLM	BUFFER	8, 16	{Chan 1-4}	(Base+U{Slot})->DW[Chan]	Shared	SM
DW	DW	R/W	A32	BUFFER	8, 16	{Chan 1-4}	(Base+Buffer+U{Slot})->DW[Chan]	GenState	SM
DW	DW	R/W	A64	BUFFER	8, 16	{Chan 1-4}	(0+U{Slot})->DW[Chan]	Canonical	SM

This message allows direct read/write accesses to untyped surfaces using 8 or 16 offsets. Up to 4 DWords are accessed beginning at the byte addresses determined. These 4 DWords correspond to the red, green, blue, and alpha channels, in that order and with red mapping to the lowest order DWord.

For BUFFER accesses, if the U offset is not DWord-aligned, the results are undefined.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Programming Note	
Context:	Untyped Surface Read/Write Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	SIMD8	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	BTS	SCRATCH,NULL	SIMD8	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	BTS	SCRATCH,NULL	SIMD8	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	BTS	SCRATCH,NULL	SIMD16	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	BTS	SCRATCH,NULL	SIMD16	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD8	{Chan 1-4}		{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	{Chan 1-4}		{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP16b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP16b_USU_SIMD16	{Chan 1-4} MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	{Chan 1-4}		{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	{Chan 1-4}		{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	{Chan 1-4}	MSD1R_US	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{Chan 1-4}	MSD1W_US	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}
R	A64	BUFFER	SIMD8	{Chan 1-4}	MSD1R_A64_US	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	{Chan 1-4}	MSD1W_A64_US	{Forbidden}	MAP64b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD16	{Chan 1-4}	MSD1R_A64_US	{Forbidden}	MAP64b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	A64	BUFFER	SIMD16	{Chan 1-4}	MSD1W_A64_US	{Forbidden}	MAP64b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}

The number of message registers in the write data payload is determined by the number of channel mask bits that are enabled.

The **Channel Mask** in the message descriptor identifies whether the corresponding channel access is disabled. For BTS surfaces, the surface state **Shader Channel Select** fields identify whether the channels are swizzled for the access.

Typed Surface Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
B	DW	R/W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	{Chan 1-4}	(Surface[U, V, R, LOD])->DW[Chan]	Surface	SG

This message allows direct read/write accesses to typed surfaces using 8 offsets. Up to 4 channels are accessed beginning at the byte address determined: red, green, blue, and alpha channels, in that order.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	1D, 2D, 3D, CUBE, BUFFER,	SIMD8	{Chan 1-4}	MSD1R_TS	{Forbidden}	MAP32b_TS_SIMD8	{Forbidden}	{Chan1-4} MDP_DW_SIMD8

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
		NULL							
W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD8	{Chan 1-4}	MSD1W_TS	{Forbidden}	MAP32b_TS_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}

The number of message registers in the write data payload is determined by the number of channel mask bits that are enabled.

The **Channel Mask** in the message descriptor identifies whether the corresponding channel access is disabled. The surface state **Shader Channel Select** fields identify whether the channels are swizzled for the access.

See Surface Formats for the description of how the data is formatted for Typed Surface read and write operations.

Atomic Operation Messages

Atomic operation messages cause atomic read-modify-write operations on the destination locations addressed. Atomic operations guarantee that no read or write to the same memory location from this thread or any other thread can occur between the read and the write. There is no guarantee on the write ordering of the individual operations in one SIMD message.

Out-of-bounds atomic accesses are dropped without writing data, and return zero for read data. Atomic accesses to pages in Tiled Resources Translation Tables are required to be IA-coherent. NULL Tiled Resources pages are handled as out-of-bounds accesses.

Untyped atomic messages use the RAW format, with surface type BUFFER or SCRATCH, and perform no type conversion. These messages use the U address parameter, which specifies the byte offset, which must be aligned on the data width (multiple of 8 for QWord, a multiple of 4 for DWord, or a multiple of 2 for Word).

Typed atomic messages use surface types SURFTYPE_1D, 2D, 3D, or BUFFER.

For atomic operations, only the red surface channel is used (the first DWord). The red surface channel select must be set to SCS_RED, in addition to all shader channel selects in the surface state following the rules for the surface format in RENDER_SURFACE_STATE.

The MDC_AOP and MDC_FOP are specified in the Atomic Operation Message Descriptor Control Fields section.

The new value of the destination is computed based on the old value of the destination, and up to two additional sources included in the message (src0 and src1). The sources used by the specific operations are described in the **MDC_AOP** and **MDC_FOP** tables. The AOP operations are subdivided into unary, binary, and trinary operand groupings. Source data payload are only sent for the binary and trinary operations.

The double-width AOP operation CMPWR_2W, also known as CMPRW8B on DWords and CMPWR16B on QWords, require the read-modify-write address offsets to be naturally aligned (QWord and OWord respectively).

The **MDC_RDC** message-specific control field controls whether a value is returned by the atomic message. The value returned depends on the message specific operation (see **MDC_AOP** and **MDC_FOP**).

Word Untyped Atomic Integer Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
W	W	R/W	BTS	BUFFER,NULL	8, 16	1	(Base+U{Slot})->W[0]	Surface	SM
W	W	R/W	SLM	BUFFER	8, 16	1	(Base+U{Slot})->W[0]	Shared	SM
W	W	R/W	A32	BUFFER	8, 16	1	(Base+Buffer+U{Slot})->W[0]	GenState	SM
W	W	R/W	A64	BUFFER	8	1	(0+U{Slot})->W[0]	Canonical	SM

This message performs atomic integer operations on untyped surfaces using 8 or 16 offsets. One Word is accessed beginning at the byte address determined.

For BUFFER accesses, each U offset must be Word-aligned.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Programming Note	
Context:	Word Untyped Atomic Integer Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	MDP_W_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	MDP_W_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_WAI		MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_WAI		MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_WAI		MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_WAI		MAP32b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_WAI		MAP32b_USU_SIMD8	{Forbidden}	MDP_W_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	SLM	BUFFER	SIMD16	1	MSD1R_WAI	{Forbidden}	MAP16b_USU_SIMD16	{Forbidden}	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP16b_USU_SIMD16	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAI		MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAI		MAP16b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP16b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_WAI		MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAI		MAP32b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAI		MAP16b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAI	{Forbidden}		MDP_AOP16_W2	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_WAI	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	MDP_W_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_WAI	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_WAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_WAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_WAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_WAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}

Programming Note	
Context:	Word Untyped Atomic Integer Messages

AOP imax/imin assume operands are signed 16-bit integers, umax/umin assume operands are unsigned integers. All other operations treat all values as 16-bit unsigned integers. Add and subtract operations wrap without any special indication.

DWord Untyped Atomic Integer Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	BTS	BUFFER,NULL	8, 16	1	(Base+U{Slot})->DW[0]	Surface	SM
DW	DW	R/W	SLM	BUFFER	8, 16	1	(Base+U{Slot})->DW[0]	Shared	SM

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	A32	BUFFER	8, 16	1	(Base+Buffer+U{Slot})->DW[0]	GenState	SM
DW	DW	R/W	A64	BUFFER	8	1	(0+U{Slot})->DW[0]	Canonical	SM

This message performs atomic integer operations on untyped surfaces using 8 or 16 offsets. One DWord is accessed beginning at the byte address determined.

For BUFFER accesses, each U offset must be DWord-aligned.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Programming Note	
Context:	DWord Untyped Atomic Integer Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP16b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP16b_USU_SIMD16	{Forbidden}	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP16b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP16b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_DWAI	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_DWAI	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_DWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_DWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_DWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_DWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}

Programming Note	
Context:	DWord Untyped Atomic Integer Messages
AOP imax/imin assume operands are signed 32-bit integers, umax/umin assume operands are unsigned integers. All other operations treat all values as 32-bit unsigned integers. Add and subtract operations wrap without any special indication.	

QWord Untyped Atomic Integer Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
QW	QW	R/W	A64	BUFFER	8	1	(0+U{Slot})->QW[0]	Canonical	SM

This message performs QWord atomic integer operations on untyped surfaces using 8 or 16 offsets. The surface format is RAW. One QWord is accessed beginning at the byte address determined.

For BUFFER accesses, each U offset must be QWord-aligned.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Programming Note	
Context:	QWord Untyped Atomic Integer Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	A64	BUFFER	SIMD8	1	MSD1R_A64_QWAI	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_QWAI	{Forbidden}	MAP64b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_QWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_QW_SIMD8	MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_QWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_QW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_QWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_A64_AOP8_QW2	MDP_QW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_QWAI	{Forbidden}	MAP64b_USU_SIMD8	MDP_A64_AOP8_QW2	{Forbidden}

Programming Note	
Context:	QWord Untyped Atomic Integer Messages
AOP imax/imin assume operands are signed 64-bit integers; umax/umin assume operands are unsigned integers. All other operations treat all values as 64-bit unsigned integers. Add and subtract operations wrap without any special indication.	

Word Untyped Atomic Float Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
W	W	R/W	BTS	BUFFER,NULL	8, 16	1	(Base+U{Slot})->W[0]	Surface	SM
W	W	R/W	SLM	BUFFER	8, 16	1	(Base+U{Slot})->W[0]	Shared	SM
W	W	R/W	A32	BUFFER	8, 16	1	(Base+Buffer+U{Slot})->W[0]	GenState	SM
W	W	R/W	A64	BUFFER	8	1	(0+U{Slot})->W[0]	Canonical	SM

This message performs atomic float operations on untyped surfaces using 8 or 16 offsets. One Word is accessed beginning at the byte address determined.

For BUFFER accesses, each U offset must be Word-aligned.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Programming Note	
Context:	Word Untyped Atomic Float Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_W2	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_WAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_WAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_WAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_WAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}

DWord Untyped Atomic Float Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	BTS	BUFFER,NULL	8, 16	1	(Base+U{Slot})->DW[0]	Surface	SM
DW	DW	R/W	SLM	BUFFER	8, 16	1	(Base+U{Slot})->DW[0]	Shared	SM
DW	DW	R/W	A32	BUFFER	8, 16	1	(Base+Buffer+U{Slot})->DW[0]	GenState	SM
DW	DW	R/W	A64	BUFFER	8	1	(0+U{Slot})->DW[0]	Canonical	SM

This message performs atomic float operations on untyped surfaces using 8 or 16 offsets. One DWord is accessed beginning at the byte address determined.

For BUFFER accesses, each U offset must be DWord-aligned.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

Programming Note	
Context:	DWord Untyped Atomic Float Messages
Optional header is not allowed.	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	BTS	BUFFER,NULL	SIMD8	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	BUFFER,NULL	SIMD8	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	BUFFER,NULL	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	BTS	BUFFER,NULL	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	SLM	BUFFER	SIMD8	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	SLM	BUFFER	SIMD8	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	SLM	BUFFER	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAF	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_DWAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_DWAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A64	BUFFER	SIMD8	1	MSD1R_A64_DWAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A64	BUFFER	SIMD8	1	MSD1W_A64_DWAF	{Forbidden}	MAP64b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}

DWord Typed Atomic Integer Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	BTS	1D, 2D, 3D, CUBE	8	1	(Surface[U, V, R, LOD])->DW[0]	Surface	SG
DW	DW	R/W	BTS	BUFFER, NULL	8	1	(Surface[U])->DW[0]	Surface	SG

This message performs atomic integer operations on typed surfaces using 8 offsets. The surface format must be one of R32_UINT, R32_SINT or R32_FLOAT, and the surface type must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, SURFTYPE_CUBE, or SURFTYPE_BUFFER. One DWord is accessed beginning at the byte address determined by the address payload and the surface format. Each access is a DWord and must be DWord-aligned.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped or return zero. The semantics depend on the address model, as listed in the above table.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1R_DWTAI	{Forbidden}	MAP32b_TS_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1W_DWTAI	{Forbidden}	MAP32b_TS_SIMD8	{Forbidden}	{Forbidden}
R	BTS	BUFFER, NULL	SIMD8	1	MSD1R_DWTAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD8	1	MSD1W_DWTAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1R_DWTAI	{Forbidden}	MAP32b_TS_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1W_DWTAI	{Forbidden}	MAP32b_TS_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER, NULL	SIMD8	1	MSD1R_DWTAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD8	1	MSD1W_DWTAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1R_DWTAI	{Forbidden}	MAP32b_TS_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	1D, 2D, 3D, CUBE	SIMD8	1	MSD1W_DWTAI	{Forbidden}	MAP32b_TS_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	BUFFER, NULL	SIMD8	1	MSD1R_DWTAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD8	1	MSD1W_DWTAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}

Programming Note	
Context:	DWord Typed Atomic Integer Messages
The U address payload specifies a pixel index into the surface, which is multiplied by the **Surface Pitch** from the Surface State to generate the byte address. That final byte address must be Dword aligned.	

Word Atomic Counter Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
W	W	R/W	BTS	Any	8	1	(Surface Append Buffer)->W[0]	Ignored	SG

This message performs atomic integer operations on the "Append Counter" associated with the surface, using 8 offsets. One Word is accessed beginning at the byte address of the surface's auxiliary buffer address, which must be Word-aligned.

For Append Counter Operations there is no address payload: the address is provided by the append counter field in the surface state. The data payloads are the same as untyped atomic integer operations. The counter values in memory are stored as untyped, so no format conversion is needed by the data port when these messages require return data to (the return data from L3 is always treated as untyped, independent of the surface format.

When accessing a surface with this message, if the Auxiliary Surface Mode of the surface state is not AUX_APPEND, the access is treated as out of bounds with the writes being ignored and the reads returning 0.

The execution mask bits may disable accesses on the corresponding SIMD slots.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	Any	SIMD8	1	MSD1R_WAC	MH1_BTS_PSM	{Forbidden}	{Forbidden}	MDP_W_SIMD8
W	BTS	Any	SIMD8	1	MSD1W_WAC	MH1_BTS_PSM	{Forbidden}	{Forbidden}	{Forbidden}
R	BTS	Any	SIMD8	1	MSD1R_WAC	MH1_BTS_PSM	{Forbidden}	MDP_W_SIMD8	MDP_W_SIMD8
W	BTS	Any	SIMD8	1	MSD1W_WAC	MH1_BTS_PSM	{Forbidden}	MDP_W_SIMD8	{Forbidden}
R	BTS	Any	SIMD16	1	MSD1R_WAC	MH1_BTS_PSM	{Forbidden}	MDP_W_SIMD16	MDP_W_SIMD16
W	BTS	Any	SIMD16	1	MSD1W_WAC	MH1_BTS_PSM	{Forbidden}	MDP_W_SIMD16	{Forbidden}

Programming Note	
Context:	Word Atomic Counter Messages

- Both unary and binary atomic integer operations require a message header.
- The atomic integer operations forbids a message header, so the Pixel Sample Mask is defaulted to be fully enabled.
- The Append Counter cannot be used with an MSAA auxiliary surface.

DWord Atomic Counter Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	BTS	Any	8	1	(Surface Append Buffer)->DW[0]	Ignored	SG

This message performs atomic integer operations on the "Append Counter" associated with the surface, using 8 offsets. One DWord is accessed beginning at the byte address of the surface's auxiliary buffer address, which must be DWord-aligned.

For Append Counter Operations there is no address payload: the address is provided by the append counter field in the surface state. The data payloads are the same as untyped atomic integer operations.

When accessing a surface with this message, if the Auxiliary Surface Mode of the surface state is not AUX_APPEND, the access is treated as out of bounds with the writes being ignored and the reads returning 0.

The execution mask bits may disable accesses on the corresponding SIMD slots.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	Any	SIMD8	1	MSD1R_DWAC	MH1_BTS_PSM	{Forbidden}	{Forbidden}	MDP_DW_SIMD8
W	BTS	Any	SIMD8	1	MSD1W_DWAC	MH1_BTS_PSM	{Forbidden}	{Forbidden}	{Forbidden}
R	BTS	Any	SIMD8	1	MSD1R_DWAC	MH1_BTS_PSM	{Forbidden}	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	Any	SIMD8	1	MSD1W_DWAC	MH1_BTS_PSM	{Forbidden}	MDP_DW_SIMD8	{Forbidden}

Programming Note	
Context:	DWord Atomic Counter Messages

- Both unary and binary atomic integer operations require a message header .
- The current implementation of the binary atomic integer operations forbids a message header, so the Pixel Sample Mask is defaulted to be fully enabled.
- The Append Counter cannot be used with an MSAA auxiliary surface.

Media Surface Read/Write Messages

The media block and transpose messages allow direct read/write accesses to media surfaces. These messages operate on rectangular blocks of pixel data.

Media surfaces are 2D types with some media-specific characteristics (for example, vertical offset handling for interlaced frames). The surface state specifies all the parameters.

There are some limitations on what data surface formats are supported by data ports, when compared to the Sampler, Render Cache, and the fixed function Video encode and decode engines. See the specific messages and Addressing 2D Media Surfaces in this volume for more details.

There are some limitations on what data surface formats are supported by data ports, when compared to the Render Cache and the fixed function Video encode and decode engines. See the specific messages and Addressing 2D Media Surfaces in this volume for more details.

Media Block Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R/W	BTS	2D, NULL	1 block	Width x Height	(Surface[X, Y])->DW[0]	Media	Ignored

The read form of this message enables a rectangular block of data samples to be read from the source surface and written into the GRF. The write form enables data from the GRF to be written to a rectangular block. The message specifies as inputs a signed (X, Y) coordinate into the 2D surface and a block size (Width, Height).

The execution mask is ignored. Out-of-bounds writes are dropped. Out-of-bounds reads return a replicated boundary pixel. More details on bounds checking is described in Addressing 2D Media Surfaces in this volume.

Applications:

Block reads/writes for media

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	2D, NULL	1 block	Height rows x Width bytes	**MSD1R_MB**	**MH_MB**	{Forbidden}	{Forbidden}	**MDP_HW1**, **MDP_HW2**, **MDP_HW4**, **MDP_HW8**
W	BTS	2D, NULL	1 block	Height rows x Width bytes	**MSD1W_MB**	**MH_MB**	{Forbidden}	**MDP_HW1**, **MDP_HW2**, **MDP_HW4**, **MDP_HW8**	{Forbidden}
R	BTS	2D, NULL	1 block	Height rows x Width bytes	**MSD_SC_MB**	**MH_MB**	{Forbidden}	{Forbidden}	**MDP_HW1**, **MDP_HW2**, **MDP_HW4**, **MDP_HW8**

The only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Accesses are allowed to SURFTYPE_NULL, reads will return 0 and writes will be ignored. Only non-IA-coherent surfaces may be used with compressed media surfaces. More details on 2D surfaces used by this message is described in Addressing 2D Media Surfaces in this volume.

See 2D Media Surface Formats in this volume for limitations on surface formats supported by this message.

Block Width and Block Height

The maximum Block Height is limited by the Block Width. The maximum data size supported in this message is 64 Dwords (256 Bytes). Block reads and writes wider than 32 bytes are supported only with either linear and Tile X surfaces.

Block Width (bytes)	Block Height (rows)	Tile Modes Supported
1-4	1-64	Linear, TileX, TileY/YF/YS
5-8	1-32	Linear, TileX, TileY/YF/YS
9-16	1-16	Linear, TileX, TileY/YF/YS
17-32	1-8	Linear, TileX, TileY/YF/YS
33-64	1-4	Linear, TileX

The layout of the read and write data payload depends on the Block Height and Block Width. The data is aligned to the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the Block Width. The figure below illustrates the layout of the rows in the registers.

Media Block Row Layout in Registers

		Width = 1-4								Width = 5-8								Width = 9-16								Width = 17-32								Width = 33-64							
Dword		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
GRF	0	8	7	6	5	4	3	2	1	4	3	2	1					2	1							1								1							
	1	16	15	14	13	12	11	10	9	8	7	6	5					4	3							2															
	2	24	23	22	21	20	19	18	17	12	11	10	9					6	5							3								2							
	3	32	31	30	29	28	27	26	25	16	15	14	13					8	7							4															
	4	40	39	38	37	36	35	34	33	20	19	18	17					10	9							5								3							
	5	48	47	46	45	44	43	42	41	24	23	22	21					12	11							6															
	6	56	55	54	53	52	51	50	49	28	27	26	25					14	13							7								4							
	7	64	63	62	61	60	59	58	57	32	31	30	29					16	15							8															

Programming Note	
Context:	Media Block Read/Write Messages

Restrictions Media Block Read and Write:

- The surface base address must be 32-byte aligned.
- Pitch must be a multiple of 64 bytes when the surface is linear.
- For YUV422 formats, the block width and offset must be pixel pair aligned (i.e. DWord-aligned).
- The block width and offset must be aligned to the size of pixels stored in the surface. For a surface with 8bpe pixels for example, the block width and offset can be byte-aligned. For a surface with 16bpe pixels, it is word-aligned.

Restrictions Media Block Write:

- For media block writes, both X Offset and Block Width must be DWord-aligned.
- Media block writes to Linear or TileX surfaces must have a height of 16 or less.
- Media block writes are not supported on compressible media surfaces.

Byte Masked Media Block Write Message

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	W	BTS	2D, NULL	1 block	Width x Height	(Surface[X, Y])->DW[0]	Media	Ignored

This message conditionally writes a rectangular block of pixels from GRF data. The message specifies as inputs a signed (X, Y) coordinate into the 2D surface with block size (Width, Height), and a Byte Mask to enable writing of the individual bytes.

The execution mask is ignored. Out-of-bounds writes are dropped. Media bounds checking is more fully described in Addressing 2D Media Surfaces in this volume.

Applications:

Block writes for media

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	2D, NULL	1 block	Height rows x Width bytes	MSD1W_MB	MH_MBBM	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8	{Forbidden}

The only surface type allowed is non-arrayed, non-mipmapped SURFTYPE_2D. Accesses are allowed to SURFTYPE_NULL, writes will be ignored. More details on 2D surfaces used by this message is described in Addressing 2D Media Surfaces in this volume.

See 2D Media Surface Formats in this volume for limitations on surface formats supported by this message.

Block Width and Block Height

The maximum Block Height is limited by the Block Width. The maximum data size supported in this message is 64 Dwords (256 Bytes).

Block Width (bytes)	Block Height (rows)	Tile Modes Supported
1-4	1-64	Linear, TileX, TileY/YF/YS
5-8	1-32	Linear, TileX, TileY/YF/YS
9-16	1-16	Linear, TileX, TileY/YF/YS
17-32	1-8	Linear, TileX, TileY/YF/YS

The data payload is aligned with the upper left pixel going into the least significant bits of the first register, and the register pitch is equal to the next power-of-2 that is greater than or equal to the Block Width. The Byte Mask applies horizontally to each row of output. When the Byte Mask bit is set, the byte is written. The figure below illustrates the layout of the row data and corresponding mask bits.

Row and Byte Mask Layout

Dword		Width = 1-4								Width = 5-8								Width = 9-16								Width = 17-32							
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
GRF	0	8	7	6	5	4	3	2	1	4	3	2	1					2	1							1							
	1	16	15	14	13	12	11	10	9	8	7	6	5					4	3							2							
	2	24	23	22	21	20	19	18	17	12	11	10	9					6	5							3							
	3	32	31	30	29	28	27	26	25	16	15	14	13					8	7							4							
	4	40	39	38	37	36	35	34	33	20	19	18	17					10	9							5							
	5	48	47	46	45	44	43	42	41	24	23	22	21					12	11							6							
	6	56	55	54	53	52	51	50	49	28	27	26	25					14	13							7							
	7	64	63	62	61	60	59	58	57	32	31	30	29					16	15							8							
Byte Mask		Bits 3:0 for each Row								Bits 7:0 for each Row								Bits 15:0 for each Row								Bits 31:0 for each Row							

Programming Note	
Context:	Byte Masked Media Block Write Message

- Linear or TileX surfaces must have a block height <= 16.
- Both X Offset and Block Width must be DWord-aligned.
- The surface base address must be 32-byte aligned.
- Pitch must be a multiple of 64 bytes when the surface is linear.
- Media block writes are not supported on compressible media surfaces.

Other Data Port Messages

This section describes the remaining data port messages that do not fit into one of the other groupings.

Read Surface Info Message

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
B	B	R	BTS	Any	1	1		Ignored	Ignored

This message is used to determine information about a surface MIP level. It returns the surface format, type, width, depth, height of the specified surface address.

The LOD is in-bounds if address payload's LOD < surface's MIPCount and if the surface's MinLOD + address payload's LOD < 15. If the LOD is not in-bounds then 0 is returned for the width, height, and depth values.

The execution mask is ignored. No bounds checking is performed on the U, V, or R address components in the address payload.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	Any	1	1	MSD_RSI	{Forbidden}	MAP32b_RSI	{Forbidden}	MDP_RSI

There is no message header for this message type.

The 64-bit Instruction Base Address returned in the Writeback Payload is specified as a 48-bit state base address and is extended to 64 bits in HW, so SW can read it for conversion of 64-bit instruction pointers.

Untyped Surface Uncompressed Write Message

Untyped Surface Uncompressed Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	W	BTS	BUFFER,NULL	8, 16	{Chan 1-4}	(Base+U{Slot})->DW[Chan]	Surface	SM

This message allows writing to untyped surfaces in uncompressed format, even though the surface mode is compressible. Surface compression mode could be either 3D or Media . If the surface is not set as compressible, then these messages are identical to regular Untyped Surface Write messages.

These messages are not permitted on SLM.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped. The semantics depend on the address model, as listed in the above table.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	BUFFER,NULL	SIMD8	{Chan 1-4}	MSD_US_UCW	{Forbidden}	MAP32b_USU_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}
W	BTS	BUFFER,NULL	SIMD16	{Chan 1-4}	MSD_US_UCW	{Forbidden}	MAP32b_USU_SIMD16	{Chan1-4} MDP_DW_SIMD16	{Forbidden}

Typed Surface Uncompressed Write Messages

Addr Align	Data Width	W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
B	DW	W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	{Chan 1-4}	(Surface[U, V, R, LOD])->DW[Chan]	Surface	SG

This message allows writing data to typed surfaces in uncompressed format, even though the surface mode is compressible. Surface compression mode could be either 3D or Media . If the surface is not set as compressible, then these messages are identical to regular Typed Surface Write messages.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped. The semantics depend on the address model, as listed in the above table.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD8	{Chan 1-4}	MSD_TS_UCW	{Forbidden}	MAP32b_TS_SIMD8	{Chan1-4} MDP_DW_SIMD8	{Forbidden}

See Surface Formats for the description of how the data is formatted for Typed Surface write operations.

Typed Surface CCS Sector Update Message

Addr Align	Data Width	W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
B	DW	W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	8	1	(Surface[U, V, R, LOD])->DW[Chan]	Surface	SG

This message allows directly updating the Compression Control Surface (CCS) state of a compressible surface, without writing any data to the surface itself. Surface's compression mode could be either 3D or Media. The operation is only permitted on compressible surfaces.

This message will update CCS state of the aligned 128B sector that includes DWs calculated by the above SIMD Address calculation. The allowed CCS update operations are defined in the **Message Descriptor field**.

The execution mask bits may disable accesses on the corresponding SIMD slots. Out-of-bounds accesses are dropped. The semantics depend on the address model, as listed in the above table.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD8	1	MSD_TS_CCS_OP	{Forbidden}	MAP32b_TS_SIMD8	{Forbidden}	{Forbidden}

See Surface Formats for the description of how the data is formatted for Typed Surface write operations.

A64 CCS Page Update Message

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
OW	DW	W	A64	BUFFER	1	1	(0+U{Slot})->DW[0]	Canonical	SM

This message allows directly updating the corresponding Compression Control Surface (CCS) state of a whole 64KB page of compressible data, without writing any data to the surface itself. This message is permitted only on addresses that are part of a compressible surface.

A64 message for CCS page update operation has single 64-bit address in the header. The corresponding CCS state for the 64KB data page that contains address is updated. The allowed CCS update operations are defined in the **Message Descriptor field**.

The address in this message must be OWord-aligned. These messages are not permitted on SLM.

Out-of-bounds accesses are dropped. The semantics depend on the address model, as listed in the above table.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	A64	BUFFER	1	1	MSD_A64_CCS_PG_OP	MH_A64_OWB	{Forbidden}	{Forbidden}	{Forbidden}

URB Read/Write Messages

The URB read and write messages allow direct read/write accesses to the Unified Return Buffer used by the 3D pipeline.

URB Fence Message

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
N/A	N/A	N/A	URB	N/A	N/A	N/A	N/A	Ignored	Ignored

A URB fence message issued by a thread causes further messages issued by the thread to be blocked until all previous URB messages have completed, or the results can be globally observed from the point of view of other threads in the system.

The execution mask is ignored. No bounds checking is performed.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
N/A	URB	N/A	N/A	N/A	MSD_URBFENCE	MH_IGNORE	{Forbidden}	{Forbidden}	MDP_DW_SIMD8

The URB fence message signals completion by returning data into the writeback register. The data returned in the writeback register is undefined. When an instruction reads the writeback register value, then this thread is blocked until all previous URB messages are globally observable. The writeback register must be read before this thread sends another data port message.

A URB fence memory is typically performed prior the thread exit message, so that the next thread dispatch that reads that URB memory will see it.

Global observability of URB memory means the URB memory is visible to all the threads that have access to that memory.

DWord URB Read/Write Messages

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
OW	DW	R/W	URB	BUFFER	8	{Chan 1-16}	Handle->OW[GlobalOffset+Offset{Slot}].DW[chan]	NA	SM

For each SIMD8 slot, this message reads or writes up to 8 or 16 contiguous DWords starting at each slot's offset.

The number of DWords to read is specified by RLEN in the message descriptor. The number of DWords to write is specified by the MLEN in the message descriptor, after accounting for the message header and address payloads.

The per-slot message offset address payload is optional. If present, it is added to the **Global Offset** in the message descriptor to calculate the offset from the handle specified in the message header.

Masked write messages supply an additional address payload: the per-slot Channel Masks that control which URB Dwords are written. The unmasked read and write messages do not supply a Channel Mask address payload and access all the channels in the message.

The execution mask bits may disable disable all channel accesses on the corresponding SIMD slots.

The bounds checks must be done to ensures only 3D shaders can access the URB destination. Access to URB will be limited to EXID = 0 and 3D shaders(Vertex, Hull, Domain and Geometry). The behavior of bounds checking will follow the same rules as Data Port bounds checking. See Bounds Checking and Faulting section for more information.

The list of FFIDs considered as 3D shaders and allowed to access the URB is expanded to include the Task (FFID_TASK) and Mesh (FFID_MESH) shaders.

Applications:

- Write 3D shader kernel results to URB for next phase of 3D pipeline
- Read 3D attributes from URB for this 3D shader kernel

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	URB	BUFFER	SIMD8	{Chan 1-8}	MSDUR_DWS	MH_URB_HANDLE	{opt} MAPU_SIMD8	{Forbidden}	{Chan 1-8} MDP_DW_SIMD8
W	URB	BUFFER	SIMD8	{Chan 1-8}	MSDUR_DWS	MH_URB_HANDLE	{opt} MAPU_SIMD8	{Chan 1-8} MDP_DW_SIMD8	{Forbidden}
W	URB	BUFFER	SIMD8	{Chan 1-8)	MSDUW_MDWS	MH_URB_HANDLE	{opt} MAPU_SIMD8 + MAPU_CMASK_SIMD8	{Chan 1-8} MDP_DW_SIMD8	{Forbidden}

Message-Specific Descriptors

All the operations supported on the 4 data ports (Data Port 0, 1, 2, and RO), and over the 4 address models (BTS, SLM, A32, A64) are described by the Message Descriptors.

All the message descriptors indicate whether a Message Header is present, and the Message Type. The decoding of each message descriptor is specific to the message type and the data port it is sent to.

Data Port 0, 1, and RO usually provide a Binding Table Index (BTI) to specify a surface in the BTS address model. The special entry 255 is used to reference Stateless A32 or A64 address model, and the special entry 254 is used to reference the SLM address model.

The special entry 251-252 is used to reference bindless resource operation. The surface is identified by the SSO (Surface State Offset) field. That field is encoded in the SENDS instruction's extended descriptor using indirect register addressing.

Entry 251 specifies the surface offset is relative to the Surface State Base Address. Entry 252 specifies the surface offset is relative to the Bindless Surface State Base Address.

Data Port Bindless Surface Extended Message Descriptor

Entries 240-250 are reserved for future use.

Data Port 0 Message Specific Descriptors

This section contains the message specific descriptors for Data Port 0.

MT_DP0 - Data Port 0 Message Types

Scattered Read/Write Messages

Message Descriptor
MSD0R_BS - Byte Scattered Read MSD
MSD0W_BS - Byte Scattered Write MSD
MSD0R_DWS - Dword Scattered Read MSD
MSD0W_DWS - Dword Scattered Write MSD

Block Read/Write Messages

Message Descriptor
MSD0R_OWB - Oword Block Read MSD
MSD0W_OWB - Oword Block Write MSD
MSD0R_OWAB - Oword Aligned Block Read MSD
MSD0R_HWB - Scratch Block Read MSD
MSD0W_HWB - Scratch Block Write MSD

Data Port 1 Message Specific Descriptors

This section contains the message specific descriptors for Data Port 1.

MT_DP1 - Data Port 1 Message Types

A64 Scattered Read/Write

Descriptor
MSD1R_A64_BS - A64 Byte Scattered Read MSD
MSD1W_A64_BS - A64 Byte Scattered Write MSD
MSD1R_A64_DWS - A64 Dword Scattered Read MSD
MSD1W_A64_DWS - A64 Dword Scattered Write MSD
MSD1R_A64_QWS - A64 Qword Scattered Read MSD
MSD1W_A64_QWS - A64 Qword Scattered Write MSD

A64 Block Read/Write

Descriptor
MSD1R_A64_OWB - A64 Oword Block Read MSD
MSD1W_A64_OWB - A64 Oword Block Write MSD
MSD1R_A64_OWUB - A64 Oword Unaligned Block Read MSD
MSD1R_A64_HWB - A64 Hword Block Read MSD
MSD1W_A64_HWB - A64 Hword Block Write MSD

Surface Read/Write

Descriptor
MSD1R_US - Untyped Surface Read MSD
MSD1W_US - Untyped Surface Write MSD
MSD1R_A64_US - A64 Untyped Surface Read MSD
MSD1W_A64_US - A64 Untyped Surface Write MSD
MSD1R_TS - Typed Surface Read MSD
MSD1W_TS - Typed Surface Write MSD

Media Messages

Descriptor
MSD1R_MB - Media Block Read MSD
MSD1W_MB - Media Block Write MSD

Read-Only Data Port Message Specific Descriptors

This section contains the message specific descriptors for the Read-Only Data Port .

MT_DP_RO - Read-Only Data Port Message Types

Block Read/WriteBlock Read/WriteBlock Read/Write

Other Messages

Message Descriptor
MSD_RSI - Read Surface Info MSD
MSD_US_UCW - Untyped Surface Uncompressed Write MSD
MSD_TS_UCW - Typed Surface Uncompressed Write MSD
MSD_US_CCS_OP - Untyped Surface CCS Operation MSD
MSD_A64_CCS_PG_OP - A64 Page CCS Operation MSD
MSD_TS_CCS_OP - Typed Surface CCS Operation MSD

URB Data Port Message Specific Descriptors

This section contains the message specific descriptors for the URB Data Port SFID_URB.

URB Messages

Message Descriptor
MSDUR_DWS - URB Dword Read
MSDUW_DWS - URB Dword Write
MSDUW_MDWS - URB Masked Dword Write

Message Headers

Message Headers provide additional parameters used by the message. When present, the Header is 1 register and specifies a common set of parameters used by most of the data port's messages. The specific combinations supported are described in the Messages section.

Data Port 0 supports various block and scattered operations for the Read/Write data cache. The Data Port 0 Message Header is programmed to provide:

Buffer Base Address	A base address offset used for A32 stateless address models. The default value is 0 when the message header is not present.
Global Offset	A Byte, DWord, or QWord offset that is added to base address offset to address the data operands. The default value is 0 when the message header is not present.
Per Thread Scratch Space	This is used for bounds checking by the data port operation, to guarantee that the address calculation does not go outside of the range expected for this thread. The default value is the maximum value (11) when the message header is not present.

The A64 Read/Write Block operation provides A64 Stateless operations on the Data Port 1 Read/Write data cache. This special Data Port 1 Message Header is programmed to provide:

Block Offset 0 **Block Offset 1**	This specifies the 1 or 2 64-bit addresses used for the A64 stateless address model.
HWord Channel Mode	This controls how channel execution masks are interpreted for HWord operations.

Surface and atomic operations, both untyped and typed, are provided on Data Port 1 for the Read/Write data cache. This Message Header is programmed to provide:

Message Header Fields for Data Port 1	
Buffer Base Address	A base address used for SLM and A32 stateless address models. The default value is 0 when the message header is not present.

Data Port 0 Message Headers

This section contains the message headers for Data Port 0.

Message Header
MH_BTS_GO - Block Message Header
MH_A32_GO - Stateless Block Message Header
MH_A32_HWB - Scratch Hword Block Message Header
MH_IGNORE - Ignored Message Header

Data Port 1 Message Headers

This section contains the message headers for Data Port 1.

Message Header
MH_A64_OWB - A64 Oword Block Message Header
MH_A64_HWB - A64 Hword Block Message Header
MH1_A32 - Stateless Surface Message Header
MH_MB - Normal Media Block Message Header
MH_MBBM - Byte Masked Media Block Message Header

URB Data Port Message Headers

This section contains the message headers for the URB Data Port.

Message Header
MH_URB_HANDLE - URB Handle Message Header

Message Address Payloads

The next sections describe all the supported formats of the Address Payloads.

Message Address Payloads are packed into registers based on the address model, surface type, and the SIMD size. The number of registers used by an address payload is 1, 2, 4, or 5.

32 Bit Address Payloads

Address Payload
MAP32b_SIMD8 - SIMD8 32-Bit Address Payload
MAP32b_SIMD16 - SIMD16 32-Bit Address Payload
MAP16b SIMD16 - SIMD16 16-Bit Address Payload

64 Bit Address Payloads

Address Payload
MAP64b_SIMD8 - SIMD8 64-Bit Address Payload
MAP64b_SIMD16 - SIMD16 64-Bit Address Payload

32 Bit Untyped Surface Address Payloads

Address Payload
MAP32b_USU_SIMD8 - SIMD8 Untyped BUFFER Surface 32-Bit Address Payload
MAP32b_USU_SIMD16 - SIMD16 Untyped BUFFER Surface 32-Bit Address Payload
MAP16b USU SIMD16 - SIMD16 Untyped BUFFER Surface 16-Bit Address Payload

64 Bit Untyped Surface Address Payloads

Address Payload
MAP64b_USU_SIMD8 - SIMD8 Untyped BUFFER Surface 64-Bit Address Payload
MAP64b_USU_SIMD16 - SIMD16 Untyped BUFFER Surface 64-Bit Address Payload

32 Bit Typed Surface Address Payloads

Address Payload
MAP32b_TS_SIMD8 - SIMD8 Typed Surface 32-Bit Address Payload
MAP32b_RSI - Read Surface Info 32-Bit Address Payload

URB Address Payloads

Address Payload
MAPU_SIMD8 - SIMD8 URB Offset Message Address Payload
MAPU_CMASK_SIMD8 - SIMD8 URB Channel Mask Message Address Payload

Message Data Payloads

The next sections describe the all the supported formats of Data Payloads.

Message Data Payloads are packed into registers based on the width of the data and on the SIMD size. The supported data payloads are 1, 2, 4, or 8 registers.

Oword Block Data Payloads

Payload
MDP_OW1L - Lower Oword Block Data Payload
MDP_OW1U - Upper Oword Block Data Payload
MDP_OW2 - Oword 2 Block Data Payload
MDP_OW4 - Oword 4 Block Data Payload
MDP_OW8 - Oword 8 Block Data Payload

Hword Block Data Payloads

Payload
MDP_HW1 - Hword 1 Block Data Payload
MDP_HW2 - Hword 2 Block Data Payload
MDP_HW4 - Hword 4 Block Data Payload
MDP_HW8 - Hword 8 Block Data Payload

Word SIMD Data Payloads

Payload
MDP_W_SIMD8 - Word SIMD8 Data Payload
MDP_W_SIMD16 - Word SIMD16 Data Payload

Dword SIMD Data Payloads

Payload
MDP_DW_SIMD8 - Dword SIMD8 Data Payload
MDP_DW_SIMD16 - Dword SIMD16 Data Payload

Qword SIMD Data Payloads

Payload
MDP_QW_SIMD8 - Qword SIMD8 Data Payload
MDP_QW_SIMD16 - Qword SIMD16 Data Payload

SIMD Atomic Operation Data Payloads

Payload
MDP_AOP8_W2 - Word SIMD8 Atomic Operation CMPWR Message Data Payload
MDP_AOP16_W2 - Word SIMD16 Atomic Operation CMPWR Message Data Payload
MDP_AOP8_DW2 - Dword SIMD8 Atomic Operation CMPWR Message Data Payload
MDP_AOP16_DW2 - Dword SIMD16 Atomic Operation CMPWR Message Data Payload
MDP_AOP8_QW1 - Qword SIMD8 Atomic Operation Return Data Message Data Payload
MDP_AOP16_QW1 - Qword SIMD16 Atomic Operation Return Data Message Data Payload
MDP_AOP8_QW2 - Qword SIMD8 Atomic Operation CMPWR8B Message Data Payload
MDP_AOP16_QW2 - Qword SIMD16 Atomic Operation CMPWR8B Message Data Payload
MDP_A64_AOP8_QW2 - Qword SIMD8 Atomic Operation CMPWR Message Data Payload
MDP_A64_AOP8_OW2 - Oword A64 SIMD8 Atomic Operation CMPWR16B Message Data Payload

Other Data Payloads

Payload
MDP_RSI - Read Surface Info Data Payload

Common Message Descriptor Controls

Many messages are related and share a common set of control fields. For example, many read and write operations use the same Message Specific Controls in their message descriptor. These common fields are described in the next sections.

Binding Table Index Message Descriptor Control Fields

Field
MDC_BTS - Surface Binding Table Index Message Descriptor Control Field
MDC_STATELESS - Stateless Binding Table Index Message Descriptor Control Field
MDC_BTS_A32 - Surface or Stateless Binding Table Index Message Descriptor Control Field
MDC_BTS_SLM_A32 - Any Binding Table Index Message Descriptor Control Field

Header Present Message Descriptor Control Fields

Field
MDC_MHP - Header Present Message Descriptor Control Field
MDC_MHR - Header Required Message Descriptor Control Field
MDC_MHF - Header Forbidden Message Descriptor Control Field
MDC_A32_MHP - A32 Scaled Header Present Message Descriptor Control Field
MDC_A64_MHP - A64 Scaled Header Present Message Descriptor Control Field

Data Blocks Message Descriptor Control Fields

Field
MDC_DB_OW - Oword Data Blocks Message Descriptor Control Field
MDC_DB_OWD - Oword Dual Data Blocks Message Descriptor Control Field
MDC_DB_HW - Hword Register Blocks Message Descriptor Control Field
MDC_A64_DB_OW - A64 Oword Data Blocks Message Descriptor Control Field
MDC_A64_DB_HW - A64 Hword Data Blocks Message Descriptor Control Field
MDC_DS - Data Size Message Descriptor Control Field
MDC_A64_DS - A64 Data Size Message Descriptor Control Field

SIMD Mode Message Descriptor Control Fields

Field
MDC_SM3 - SIMD Mode 3 Message Descriptor Control Field
MDC_SM3S - Subset SIMD Mode 3 Message Descriptor Control Field
MDC_SM2 - SIMD Mode 2 Message Descriptor Control Field
MDC_SM2R - Reversed SIMD Mode 2 Message Descriptor Control Field
MDC_SM2S - Subset SIMD Mode 2 Message Descriptor Control Field
MDC_SG3 - Slot Group 3 Message Descriptor Control Field
MDC_SG2 - Slot Group 2 Message Descriptor Control Field

Atomic Operation Message Descriptor Control Fields

Field
MDC_AOP1 - Atomic Integer Unary Operation Message Descriptor Control Field
MDC_AOP2 - Atomic Integer Binary Operation Message Descriptor Control Field
MDC_AOP3 - Atomic Integer Ternary Operation Message Descriptor Control Field
MDC_AOP3S - Subset Atomic Integer Ternary Operation Message Descriptor Control Field
MDC_FOP2 - Atomic Float Binary Operation Message Descriptor Control Field
MDC_FOP3 - Atomic Float Ternary Operation Message Descriptor Control Field

Other Message Descriptor Control Fields

Field
MDC_IAR - Invalidate After Read Message Descriptor Control Field
MDC_CMODE - Channel Mode Message Descriptor Control Field
MDC_CMASK - Channel Mask Message Descriptor Control Field
MDC_UW_CMASK - Untyped Write Channel Mask Message Descriptor Control Field
MDC_VLSO - Vertical Line Stride Override Message Descriptor Control Field

Common Message Payload Controls

Some message headers and payloads are related and share a common set of control fields. These common payload fields are described below.

Header Controls

Control
MHC_FFTID - FFTID Message Header Control
MHC_A32_BBA - A32 Buffer Base Address Message Header Control
MHC_A64_CMODE - Hword Channel Mode Message Header Control
MHC_BDIM - Block Dimensions Message Header Control
MHC_MB_CONTROL - Normal Media Block Message Header Control
MHC_MBBM_CONTROL - Byte Masked Media Block Message Header Control
MHC_PTSS - Per Thread Scratch Space Message Header Control

Address Controls

Control
MACR_32b - SIMD 32-Bit Address Payload Control
MACR_64b - SIMD 64-Bit Address Payload Control
MACD_LOD - LOD Message Address Payload Control
MACR_LOD_SIMD8 - SIMD8 LOD Message Address Payload Control
MACD_MSAA_SN - MSAA Sample Number Message Address Control

Data Controls

Control
MDCR_W - Word Data Payload Register
MDCR_DW - Dword Data Payload Register
MDCR_QW - Qword Data Payload Register
MDCR_OW - Oword Data Payload Register

URB Controls

Control
MACD_URB_CMASK - URB Channel Mask Payload Control
MHC_URB_HANDLE - URB Handle Message Header Control

New Data Port Messages

A new collection of data port messages that support load, store, and atomic operations.

Each data port uses a different Shared Function ID (SFID).

TGM	Typed Global Memory
SLM	Shared Local Memory
UGM	Untyped Global Memory

The old data port messages are still present for compatibility. They are translated by HW to the new messages.

Messages

The dataport messages are load, store, atomic and ccs update operations. Most of the messages are supported on the untyped global (*ugm*), utyped low-priority global (*ugml*), typed global (*tgm*), and shared local memory (*slm*) ports.

The operations support a collection of data sizes, address sizes, address types, and surface types. The table below is a summary of the messages supported. For exact programming restrictions on each of the message combinations shown in the table, please refer to the per-message BXML pages.

For typed messages ("Port" == *tgm*), the "Data Size" column indicates the data size in memory, and not the data size in GRF. Typed message's data size is based on the element format specified in the surface state.

"SIMT Mask" column shows the allowed static mask values that can be programmed in the *send* instruction. Some data size/vector size combinations are further limited to a maximum SIMT size of 16, as detailed in the individual Message's BXML.

"Surface Types" column: All types other than "Flat" and "Stateless" are stateful. "Stateless" implies a BTI value of 255. For stateful messages, the supported surface types from the surface state is shown under the "Surface Type". E.g. "Buffer" implies the "surface type" programmed in the message's Surface State is SURFTYPE_BUFFER. "4D" implies arrayed 3D surface.

Programming Note
For data-port TGM, only the fence message is supported. Legacy encoding must be used for all non-fence Typed messages.

Message	Port	Data Size	Addr Size	Addr Alignment	Vector Size	Transpose	SIMT Mask	Surface Types	Data Types
load store	ugm	D8U32, D16U32, D32, D64	A32, A64	Byte	1	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Scratch, Null	Untyped, Raw format
	ugm	D32, D64	A32, A64	Data size	2, 3, 4, 8	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Scratch, Null	Untyped, Raw format
	ugm	D32, D64	A32, A64	Data size	1, 2, 3, 4, 8, 16, 32, 64	On	1	Flat, Stateless, Buffer, Scratch, Null	Untyped, Raw format
	slm	D8U32, D16U32, D32, D64	A16, A32	Byte	1	Off	1,2,4,8,16,32	Flat	Untyped
	slm	D32, D64	A16, A32	Data size	2, 3, 4, 8	Off	1,2,4,8,16,32	Flat	Untyped
	slm	D32, D64	A16, A32	Data size	1, 2, 3, 4, 8, 16, 32, 64	On	1	Flat	Untyped
load store	ugml	D8U32, D16U32, D32, D64	A32, A64	Byte	1	Off	1,2,4,8,16,32	Flat, Stateless, Buffer	Untyped, Raw format
	ugml	D32, D64	A32, A64	Data size	2, 4	Off	1,2,4,8,16,32	Flat, Stateless, Buffer	Untyped, Raw format
	ugml	D32, D64	A32, A64	Data size	2, 3, 4, 8, 16, 32, 64	On	1	Flat, Stateless, Buffer	Untyped, Raw format
store_uncompressed	ugm	D8U32, D16U32, D32, D64	A32	Byte	1	Off	1,2,4,8,16,32	Buffer, Null	Untyped, Raw format
	ugm	D32, D64	A32	Data size	2, 3, 4, 8	Off	1,2,4,8,16,32	Buffer, Null	Untyped, Raw format
	ugm	D32, D64	A32	Data size	1, 2, 3, 4, 8, 16, 32, 64	On	1	Buffer, Null	Untyped, Raw format
store_uncompressed_cmask	tgm	D8, D16, D32, D64, D128	A32	Data size	1, 2, 3, 4	Off	1,2,4,8,16,32	Buffer, 1D, 2D, 3D, 4D, Null	Surface format

Message	Port	Data Size	Addr Size	Addr Alignment	Vector Size	Transpose	SIMT Mask	Surface Types	Data Types
load_cmask store_cmask	ugm	D32	A32, A64	Dword	1, 2, 3, 4	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Scratch, Null	Untyped, Raw format
	ugml	D32	A32, A64	Dword	1, 2, 3, 4	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Null	Untyped, Raw format
	slm	D32	A16, A32	Dword	1, 2, 3, 4	Off	1,2,4,8,16,32	Flat	Untyped
	tgm	D8, D16, D32, D64, D128	A32	Data size	1, 2, 3, 4	Off	1,2,4,8,16,32	Buffer, 1D, 2D, 3D, 4D, Null	Surface format
	ugm	D8, D16, D32, D64	(Block.X, Block,Y)	Data size	(Width, Height)	Off	1	2D	Untyped, Raw format
atomic_inc atomic_dec atomic_add atomic_sub atomic_min atomic_max atomic_umin atomic_umax atomic_cmpxchg atomic_and atomic_or atomic_xor atomic_load atomic_store	ugm	D16U32, D32, D64	A32, A64	Data size	1	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Scratch, Null	Untyped, Raw format
	ugml	D16U32, D32, D64	A32, A64	Data size	1	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Null	Untyped, Raw format
	slm	D16U32, D32	A16, A32	Data size	1	Off	1,2,4,8,16,32	Flat	Untyped
	tgm	D32, D64	A32	Data size	1	Off	1,2,4,8,16,32	Buffer, 1D, 2D, 3D, 4D, Null	Restricted surface format
	tgm	D16	A32	Data size	1	Off	1,2,4,8,16,32	Buffer, 1D, 2D, 3D, 4D, Null	Restricted surface format
atomic_fadd atomic_fsub atomic_fmin atomic_fmax atomic_fcmpxchng	ugm	D32, D64	A32, A64	Data size	1	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Scratch, Null	Untyped, Raw format
	ugml	D32, D64	A32, A64	Data size	1	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Null	Untyped, Raw format

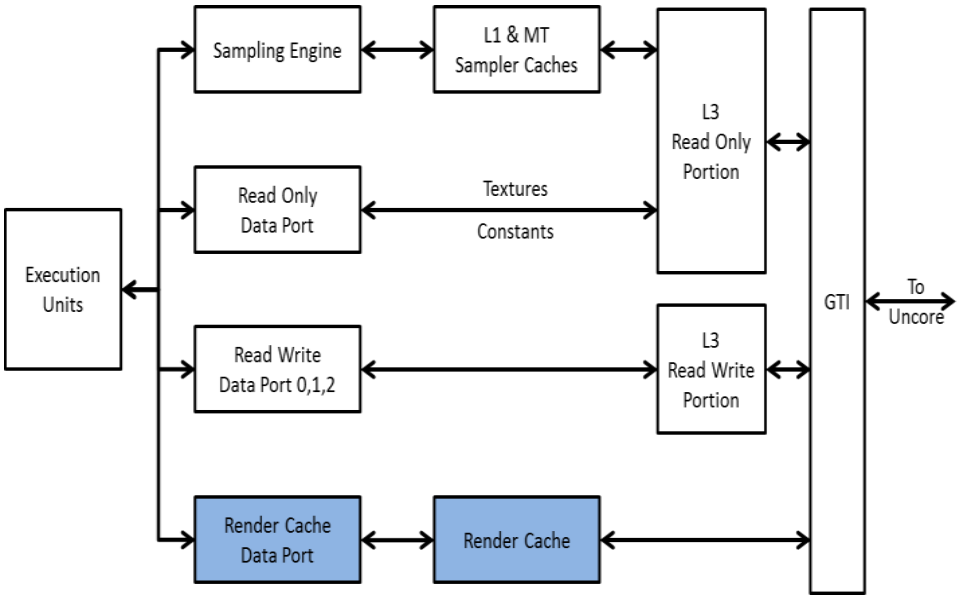
Message	Port	Data Size	Addr Size	Addr Alignment	Vector Size	Transpose	SIMT Mask	Surface Types	Data Types
fence	ugm	n/a			n/a	n/a	n/a	n/a	n/a
	ugml	n/a			n/a	n/a	n/a	n/a	n/a
	slm	n/a			n/a	n/a	n/a	n/a	n/a
	tgm	n/a			n/a	n/a	n/a	n/a	n/a
load_status	ugm	D8U32, D16U32, D32, D64	A32, A64	Byte	1	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Null	Untyped, Raw format
	ugm	D32, D64	A32, A64	Data size	2, 3, 4, 8	Off	1,2,4,8,16,32	Flat, Stateless, Buffer, Null	Untyped, Raw format
	tgm	D8, D16, D32, D64, D128	A32	Data size	1, 2, 3, 4	Off	1,2,4,8,16,32	Buffer, 1D, 2D, 3D, 4D	Surface format
ccs_sec_uncompress	tgm	D8, D16, D32, D64, D128	A32	Data size	1, 2, 3, 4	Off	1,2,4,8,16,32	Buffer, 1D, 2D, 3D, 4D, Null	Surface format
ccs_sec_clear	tgm	D8, D16, D32, D64, D128	A32	Data size	1, 2, 3, 4,	Off	1,2,4,8,16,32	Buffer, 1D, 2D, 3D, 4D, Null	Surface format
ccs_page_clear, ccs_page_uncompress	ugm	n/a	A64	64KB	1	Off	1	Flat	Untyped
read_state_info	tgm	n/a	n/a	n/a	n/a	n/a	1	Buffer, 1D, 2D, 3D, 4D, Null	n/a

Pixel Data Port

The pixel data port allows read/write accesses to render targets. Render target is cached in Render Color Cache (RCC). Auxiliary Surface (a.k.a Control Surface) i.e. MCS buffer corresponding to a render target is cached in MSC. Messages on O-bus with Shared Function ID = Render Target (RCC) are routed to Pixel Dataport.

The diagram below shows how the Pixel Data Port connects with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.

Pixel Data Port Connection to Memory



Render Target

This section describes the Render Target as pixel output from the Pixel Shader stage in the 3D Pipeline.

End of Thread Usage

Render Target data port messages may not have the End of Thread bit set in the message descriptor other than the following exceptions:

The Render Target Write message may have End of Thread set for pixel shader threads dispatched by the windower in non-contiguous dispatch mode.

Programming Note	
Context:	End of Thread Usage
When Dispatch Rate is Coarse and the Shader converts to Pixel Phases, the EOT message is required with masks being zero after all the phases.	

Bounds Checking

Read and write accesses to pixels outside of the surface are considered out-of-bounds. However, if only the **Render Target Array Index** is out of bounds and the rest of the access is in-bounds, then the index is set to zero and the read or write surface access is considered in-bounds.

Writes that are out-of-bounds are suppressed and do not modify memory contents. If an in-bounds pixel component is missing due to the Surface Format, the write is suppressed. If an in-bounds pixel component is masked off (Pixel Sample Mask bit is zero), the write is suppressed.

Reads that are out-of-bounds are suppressed and return zero for all components. If an in-bounds pixel component is missing due to the Surface Format, a zero is returned for RGB and a 1.0 is returned for

Alpha. A surface read is performed on any in-bounds pixel where the component is present in the Surface Format, regardless of whether it is masked off (Pixel Sample Mask bit is zero). All render target reads, in-bounds or out-of-bounds, modify the GRF.

Render Target Fast Clear

Fast clear of the render target is performed by setting the **Render Target Fast Clear Enable** field in **3DSTATE_PS** and rendering a rectangle The size of the rectangle is related to the size of the MCS.

The following is required when performing a render target fast clear:

- The render target(s) is/are bound as they normally would be, with the MCS surface defined in SURFACE_STATE.
- A rectangle primitive of the same size as the MCS surface is delivered. Please make sure to read MCS/CCS Buffers for Render Targets page for clearing rectangle size guidelines.
- **Depth Test Enable, Depth Buffer Write Enable, Stencil Test Enable, Stencil Buffer Write Enable,** and **Alpha Test Enable** must all be disabled.
- Pixel Shader is not required for fast clearing or resolving render target.

Color Fast Clear Synchronization

Preamble pre fast clear synchronization	Postamble post fast clear synchronization	Additional global programming
PIPE_CONTROL: PS sync stall = 1 Tile Cache Flush = 1 RT Write Flush = 1 HDC Flush = 1 DC Flush = 1 SW may choose Range Flush instead of Tile Cache Flush for performance reasons Objective of the preamble flushes is to ensure all data is evicted from L1 caches prior to fast clear.	**PIPE_CONTROL**: PS sync stall = 1 RT flush = 1	Wait on Depth Stall Done Enable = 1

Programming Note	
Context:	Workaround
SW must try one of the following two options 1. Before fast clearing any resource, SW must partially resolve the resource i.e. corresponding CCS for the resource MUST NOT be in CLEAR state 2. Clear Color address is based on clear color used for clear DDI	

Programming Note	
Context:	Render Target Fast Clear
SW needs to disable Render Target Fast clear for surface type = 2D, surface format = 8 bpp, tile format = TYS pr TY, Mip is not aligned to 32x4 pixels	

HwManaged FastClear allows SW to store FastClearValue in separate graphics allocation, instead of keeping them in RENDER_SURFACE_STATE. This behavior can be enabled by setting **ClearValueAddressEnable** in RENDER_SURFACE_STATE.

Proper sequence of commands is as follows:

1. Storing clear color to allocation.

2. Ensuring that step 1. is finished and visible for TextureCache.

3. Performing FastClear.

Step 2. is required on products with ClearColorConversion feature. This feature is enabled by setting **ClearColorConversionEnable.** This causes HW to read stored color from ClearColorAllocation and write back with the native format or RenderTarget - and clear color needs to be present and visible. Reading is done from TextureCache, writing is done to RenderCache.

How to handle steps 1&2 properly:

If ClearValueAllocation was previously read (either with sampler reading this surface with **ClearValueAddressEnable** or with FastClear operation using Clear Color Conversion feature) SW must invalidate TextureCache with proper PIPE_CONTROL. When certain L3/Tile Cache configuration are used, there might be also TileCacheFlush required.

SW must store clear color using MI_STORE_DATA_IMM with **ForceWriteCompletionCheck** bit set.

Render Target Resolve

If the MCS is enabled on a non-multisampled render target, the render target must be resolved before being used for other purposes (display, texture, CPU lock) The clear value from SURFACE_STATE is written into pixels in the render target indicated as clear in the MCS. This is done by setting the **Render Target Resolve Enable** field in 3DSTATE_PS and rendering a full render target sized rectangle. Once this is complete, the render target will contain the same contents as it would have had the rendering been performed with MCS surface disabled. In a typical usage model, the render target(s) need to be resolved after rendering and before using it as a source for any consecutive operation.

When performing a render target resolve, PIPE_CONTROL with end of pipe sync must be delivered.

The Resolve Rectangle size is same as Clear Rectangle size

Programming Note	
Context:	Render Target Resolve
• **Depth Test Enable, Depth Buffer Write Enable, Stencil Test Enable, Stencil Buffer Write Enable,** and **Alpha Test Enable** must all be disabled. • This render target resolve procedure is not supported on multisampled render targets. Unresolved multisampled render targets are directly supported by the sampling engine, which resolves clear values in addition to decompressing the surface This applies to both *ld2dms* and *sample2dms* messages.	

Execution Mask

For Render Target Write messages, either the message header **Pixel Sample Mask** or the channel execution mask is used to control the pixel write operations. If the Message Header is present, then the **Pixel Sample Mask** is used and the channel execution mask is ignored. If the Message Header is not present, then the channel execution mask is used in place of the **Pixel Sample Mask**.

For Render Target Read messages, the channel execution mask is used to control GRF write operations. The message returns zeroes for samples not covered by current pixel coverage mask or samples outside of MSAA index range.

Flushing the Render Cache

The render cache can be flushed via PIPE_CONTROL command with "Render Target Cache Flush" bit set. Typically, end of pipe synchronization that requires render targets to be made visible to Sampler or CPU or Display should use this mechanism to flush render cache. With the Render Target Cache Flush feature, some cases do require flushing render cache as mentioned in the respective feature's sections.

Multiple Render Targets (MRT)

Pixel Shader can output more than one color. Each color is accessed with a separate Render Target Write message, each with a different surface indicated (different binding table index). In this Multiple Render Target (MRT) case, depth buffer and stencil buffer are updated only by the message(s) to the last render target, indicated by the Last Render Target Select bit set to clear the pixel scoreboard bits.

MRT is not supported when one or more RTs have these surface formats: YCRCB_SWAPUVY, YCRCB_SWAPUV, YCRCB_SWAPY, or YCRCB_NORMAL.

MCS/CCS Buffers for Render Target(s)

Lossless Color Compression on Render target can be enabled for the purposes described below:

1. MMIO bit Cache Mode 1 (0x7004) register bit 5
2. MMIO bit Cache Mode 1 (0x7004) register bit 15
3. RT surface state (Auxiliary Surface Mode[2:0])

Note: Lossless Color Compression can only be applied to Surfaces which are Linear, Tile4, or Tile64. (TileY/TileYF/TileYS on older devices)

Fast Clear does not support Linear color surfaces.

The following table summarizes modes of operation related to the Lossless Color Compression on Render target:

Cache Mode 1 MMIO Bit 5 (Please refer to Vol 1c)	Auxiliary Surface Mode Surface State	Cache Mode 1 MMIO Bit 15 (Please refer to Vol 1c)	Operation
1	X	X	Normal mode of operation i.e. no MSAA compression and no color clear
0	AUX_NONE (0h)	X	Normal mode of operation i.e. no MSAA compression and no color clear
0	AUX_CCS_D (1h)	X	Depending on the Number of multi-samples, either MCS or CCS is enabled. No MSAA or Color Compression is enabled.
0	AUX_CCS_E (5h)	1	Depending on the Number of multi-samples, either MCS or CCS is enabled. For Number of multi-samples > 1 MSAA Compression is enabled. For Number of multi-samples = 1 Color Compression is disabled.
0	AUX_CCS_E (5h)	0	Depending on the Number of multi-samples, either MCS or CCS is enabled. For Number of multi-samples > 1 MSAA Compression is enabled. For Number of multi-samples = 1 Color Compression is enabled.
0	AUX_MCS_LCE	0	For Number of multi-samples > 1, MSAA compression and Color Compression, both are enabled. For, this case, SW must allocate both CCS and MCS. CCS is per each plane of the multi-sample PLANAR surface.

MSAA	Width of Clear Rect	Height of Clear Rect
2x	Ceil(1/8*width)	Ceil(1/2*height)
4X	Ceil(1/8*width)	Ceil(1/2*height)
8X	Ceil(1/2*width)	Ceil(1/2*height)
16X	width	Ceil(1/2*height)

- **MSAA Compression:** Multi-sample render target is bound to the pipeline and MSAA compression feature is enabled. In this case, MCS buffer stores the information required for MSAA compression algorithm. The size and layout of the MCS buffer is based on per-pixel RT. For 4X and 8X MSAA, MCS buffer element is 8bpp and 32bpp respectively. Height, width, and layout of MCS buffer in this case must match with Render Target height, width, and layout. MCS buffer is tiledY. When MCS buffer is enabled and bound to MSRT, it is required that it is cleared prior to any rendering. A clear value can be specified optionally in the surface state of the corresponding RT. Clear pass for this case requires that scaled down primitive is sent down with upper left coordinate to coincide with actual rectangle being cleared. For MSAA, clear rectangle's height and width need to as show in the following table in terms of (width, height) of the RT. MCS Buffer only Supports Tile4 mode.
- **Fast Color Clear:** This feature has a dedicated page. When non multi-sample render target is bound to the pipeline and MCS buffer is enabled, MCS buffer is used as an intermediate (coarse granular) buffer per RT. Hence, MCS buffer is used to improve render target clear. When MCS is buffer is used for color clear of non-multisampler render target, the following restrictions apply:

Color Clear of Non-MultiSampler Render Target Restrictions

<table>
<tr><th>Restrictions</th></tr>
<tr><td>Support is limited to tiled render targets.</td></tr>
<tr><td>They are supported with MCS buffer layout with these alignments in the RT space: Horizontal Alignment = 128 and Vertical Alignment = 64.</td></tr>
<tr><td>MCS and Lossless compression is supported for TiledY/TileYs/TileYf non-MSRTs only.</td></tr>
<tr><td>Clear is supported only on the full RT; i.e., no partial clear or overlapping clears.</td></tr>
<tr><td>Fast clear to 0 is not supported for MSAA > 1x</td></tr>
<tr><td>The following table describes the RT alignment:
<table>
<tr><th>TiledY RT CL</th><th>Pixels</th><th>Lines</th></tr>
<tr><td>bpp</td><td></td><td></td></tr>
<tr><td>8</td><td>32</td><td>4</td></tr>
<tr><td>16</td><td>16</td><td>4</td></tr>
<tr><td>32</td><td>8</td><td>4</td></tr>
<tr><td>64</td><td>4</td><td>4</td></tr>
<tr><td>128</td><td>2</td><td>4</td></tr>
<tr><th>TiledX RT CL</th><th></th><th></th></tr>
<tr><td>bpp</td><td></td><td></td></tr>
<tr><td>32</td><td>16</td><td>2</td></tr>
<tr><td>64</td><td>8</td><td>2</td></tr>
<tr><td>128</td><td>4</td><td>2</td></tr>
</table></td></tr>
<tr><td>CCS buffer (non-MSRT) is supported only for RT formats 8bpp, 16bpp, 32bpp, 64bpp, and 128bpp.</td></tr>
</table>

Restrictions
Clear pass must have a clear rectangle that must follow alignment rules in terms of pixels and lines as shown in the table below. Further, the clear-rectangle height and width must be multiple of the following dimensions. If the height and width of the render target being cleared do not meet these requirements, an MCS buffer can be created such that it follows the requirement and covers the RT.

Programming Note	
Context:	CCS surface initialization
CCS surface does not require initialization. Illegal CCS vales are treated as uncompressed memory. Surface with compression enabled may be written to without prior fast clear call.	

RT Pixel Blocks Per CCS CL: This is supposed to be used for sizing the CCS Surface.

Tile-YF RT	Pixels	Lines
bpp		
8	256	64
16	256	32
32	128	32
64	128	16
128	64	16

Tile-YS RT	Pixels	Lines
bpp		
8	128	128
16	128	64
32	64	64
64	64	32
128	32	32

Tile-Y RT	Pixels	Lines
bpp		
8	512	32
16	256	32
32	128	32
64	64	32
128	32	32

Two basic changes in the control surface mapping cause new scaling requirements with clear and resolve operations. These changes are: 1) introduction of Tile4 and Tile64 as new tiling formats (replacing TileF and TileS respectively) 2) CCS layout is memory hash aware, CCS is a flat monolithic structure.

With the above changes, fast clear and resolve rectangles need to be scaled based on 64KB aligned surface for Tile64 surfaces. HW has a burden to use the actual surface parameter to actually fragment the portion of the surface that's not aligned to 64KB and clear them appropriately.

To optimize the performance CCS buffer (when bound to 1X RT) clear similarly to CCS buffer clear for MSRT case, clear rect is required to be scaled by the following factors in the horizontal and vertical directions:

CCS CL for Tile4 RCC	Horizontal Scale Down Factor	Vertical Scale Down Factor
bpp		
8	1024	16
16	512	16
32	256	16
64	128	16
128	64	16

CCS CL for Tile64	Horizontal Scale Down Factor	Vertical Scale Down Factor
bpp		
8	128	128
16	128	64
32	64	64
64	64	32
128	32	32

SW must ensure that clearing rectangle dimensions cover the entire area desired, to accomplish this task initial X/Y dimensions need to be rounded up to next multiple of scaledown factor before dividing by scale down factor:

clear_rect_X_size = (X + (X % scale_X == 0 ? 0 : scale_X)) / scale_X

clear_rect_Y_size = (Y + (Y % scale_Y == 0 ? 0 : scale_Y)) / scale_Y

or simply:

clear_rect_X_size = Ceiling(X / scale_X)

clear_rext_Y_size = Ceiling(Y / scale_Y)

X,Y are in units of pixels

The following SW requirements for MCS buffer clear functionality apply in addition to the general SW requirements listed below:

- For non-MSRTs, loss less compression of render targets is supported for 32, 64, and 128 bpp surfaces as described in the Shared Functions Data Port under Render Target Write. Lossless RT compression can be enabled for each RT surface with auxiliary surface control bits as described in the Surface State. This feature changes the MSC layout for various tiled and BPP formats as

described in above sections. Fast clear, render, and resolve operations are fundamentally the same. Since Sampler support reading this auxiliary (MCS) buffer for non-MSRTs, resolve passes can be avoided in the cases when fast cleared and possibly compressed RTs are consumed by the sampler.

- SW does not need to compile any PS for clear and resolve passes but must ensure that PS dispatch enable bit is set.

Any transition from any value in {Clear, Render, Resolve} to a different value in {Clear, Render, Resolve} requires end of pipe synchronization.

The following are the general SW requirements for MCS buffer clear functionality:

- At the time of Render Target creation, SW needs to create clear-buffer, i.e., MCS buffer.
- At the clear time, clear value for that RT must be programmed and clear enable bit must be set in the surface state of the corresponding RT.
- SW must clear the RT with setting a RT clear bit set in the PS state during the clear pass as described in the following sub-section.
- Since only one RT is bound with a clear pass, only one RT can be cleared at a time. To clear multiple RTs, multiple clear passes are required.
- If Software wants to enable Color Compression without Fast clear, Software needs to initialize MCS with zeros.
- Lossless compression and CCS initialized to all F (using HW Fast Clear or SW direct Clear) on the same surface is not supported.
- Before binding the "cleared" RT to texture OR honoring a CPU lock OR submitting for flip, SW must ensure a resolve pass. Such a resolve pass is described in the following sub-section.

CCS is a linear buffer created for storing meta-data (AUX data) for lossless compression. This buffer related information is mentioned in Render Surface State. CCS buffer's size is based on the padded main surface (after following Halign and Valign requirements mentioned in the Render Surface State).
CCS_Buffer_Size = Padded_Main_Surface_Size/256

Programming Note	
Context:	Full Surface 3D/Volumetric Fast Clear

Following SW programming optimization is used to speed up full surface Volumetric/3D texture Fast Clear.

It is useful when clearing entire surface, all depth slices, all mip's, to one value.

We re-describe original Volumetric/3D surface as 2D surface encompassing all the original sub resources, than we can clear it using single 2D Fast Clear draw call.

This is accomplished by following calculation that produces new 2D Surface Width x 2D Surface Height that is used to program Render Surface State.

Inputs:

- Format Bits per pixel
- 3D Surface Width
- 3D Surface QPitch
- 3D Surface Depth

Outputs:

- 2D Surface Width
- 2D Surface Height

Assumptions:

- Surface uses Tile64, and is aligned to 64kb page boundaries
- We are clearing all the 3D surface sub resources (all slices, all mips)
- This method of clearing works on Mip Mapped and non Mip Mapped 3D surfaces

2D Surface Size Calculation formula:

Start with picking appropriate parameters based on format bits per pixel:

Bits per pixel	2D Page* Width	2D Page Height	3D Page Width	3D Page Height	3D Page Depth
8	256	256	64	32	32
16	256	128	32	32	32
32	128	128	32	32	16
64	128	64	32	16	16
128	64	64	16	16	16

Programming Note	
Context:	Full Surface 3D/Volumetric Fast Clear

*Page refers to 64kb page in Tile64 mode

All the calculations are performed in integer math:

Horizontal Scale = 2D Page Width / 3D Page Width

Vertical Scale = 2D Page Height / 3D Page Height

*2D Surface Width = Align(3D Surface Width, 3D Page Width) * Horizontal Scale*

*2D Surface Height = Align(3D Surface QPitch, 3D Page Height) * Vertical Scale * Align(3D Surface Depth, 3D Page Depth) / 3D Page Depth*

Align function aligns input parameter to next multiple of alignment parameter:

```
int Align(int Input, int Alignment)
{
    return ((Input / Alignment) * Alignment) + ((Input % Alignment) ? Alignment : 0);
}
```

Resulting 2D surface can be cleared using single 2D Fast Clear, use regular 2D Fast Clear instructions from this point.

See 2D Fast Clear section above.

Accessing Render Target

The final results of pixel shaders are written to either UAV surfaces or to Render target surfaces. Render targets support a large set of surface formats (refer to *Render Target Surfaces* for details) with hardware conversion from the format delivered by the thread. The render target message processing in HW includes format conversion, alpha-test, alpha-to-coverage, depth/stencil tests, blending and logop depending on various states.

The render target read/write messages are specifically for the use of pixel shader threads spawned by the Pixel Shader Dispatch(PSD), and may not be used by any other thread types.

Message Common Control Fields

The following data tables describe common control fields used in the Render Target message descriptors.

MDC_RT_SGS - Slot Group Select Render Cache Message Descriptor Control Field

Message Headers

The render target messages use a two-register message header.

If the header is not present, the behavior is as if the message was sent with most fields set to the same value that was delivered in R0 and R1 on the pixel shader thread dispatch. The following fields, which are not delivered in the pixel shader dispatch, behave as if they are set to zero:

Render Target Index

Source0 Alpha Present to Render Target

Additionally, **Render Target Read** messages must include a header.

MH_RT - Render Target Message Header

Render Target Specific Extended Message Descriptor

All Render Target Read and Write messages use same format of Extended Message Descriptor:

Extended Message Descriptor Render Target

Render Target Message Specific Descriptors

This section contains various registers for message specific descriptors.

MT_DP_RT - Render Data Port Message Types

MSD_RTW_SIMD16 - SIMD16 Render Target Write MSD

MSD_RTW_REP16 - REP16 Render Target Write MSD

MSD_RTW_HI8DS - HI8DS Render Target Write MSD

MSD_RTW_LO8DS - LO8DS Render Target Write MSD

MSD_RTW_SIMD8 - SIMD8 Render Target Write MSD

MSD_RTR_SIMD16 - SIMD16 Render Target Read MSD

MSD_RTR_SIMD8 - SIMD8 Render Target Read MSD

MSD_RTWH_SIMD16 - Half Precision SIMD16 Render Target Write MSD

MSD_RTWH_REP16 - Half Precision REP16 Render Target Write MSD

MSD_RTWH_HI8DS - Half Precision HI8DS Render Target Write MSD

MSD_RTWH_LO8DS - Half Precision LO8DS Render Target Write MSD

MSD_RTWH_SIMD8 - Half Precision SIMD8 Render Target Write MSD

Render Target Surfaces

This section contains information on render target surface types and surface formats.

Render Target Surface Types

All surface types, except SURFTYPE_SCRATCH, are allowed.

For SURFTYPE_BUFFER and SURFTYPE_1D surfaces, only the X coordinate is used to index into the surface. The Y coordinate must be zero.

For SURFTYPE_1D, 2D, 3D, and CUBE surfaces, a Render Target Array Index is included in the input message to provide an additional coordinate. The Render Target Array Index must be zero for SURFTYPE_BUFFER.

The surface format is restricted to the set supported as render target. If source/dest color blend is enabled on Render Target Write message, the surface format is further restricted to the set supported as alpha blend render target.

Cannot be used on a surface in field mode (Vertical Line Stride = 1).

Render Target Write Surface Formats

The following table indicates the surface formats supported by the Render Target Write message. All Render Target formats supported for Render Target Writes are also supported for Render Target Reads, except YCRCB formats.

The table lists the Surface Formats supported for reading and/or writing of Render Targets. For blending operations, the named Projects support color blending on the listed Surface Formats.

Surface Formats for Render Target Messages

Surface Format Name	Color Blend Support	Message Type Support	Lossless Compression Support
R32G32B32A32_FLOAT	All	Read/Write	All
R32G32B32A32_SINT	*Not Supported*	Read/Write	All
R32G32B32A32_UINT	*Not Supported*	Read/Write	All
R32G32B32X32_FLOAT	All	Read/Write	All
R16G16B16A16_UNORM	All	Read/Write	All
R16G16B16A16_SNORM	All	Read/Write	All
R16G16B16A16_SINT	*Not Supported*	Read/Write	All
R16G16B16A16_UINT	*Not Supported*	Read/Write	All
R16G16B16A16_FLOAT	All	Read/Write	All
R32G32_FLOAT	All	Read/Write	All
R32G32_SINT	*Not Supported*	Read/Write	All
R32G32_UINT	*Not Supported*	Read/Write	All
R16G16B16X16_FLOAT	All	Read/Write	All
B8G8R8A8_UNORM	All	Read/Write	All
B8G8R8A8_UNORM_SRGB	All	Read/Write	All
R10G10B10A2_UNORM	All	Read/Write	All
R10G10B10A2_UINT	*Not Supported*	Read/Write	All

Surface Format Name	Color Blend Support	Message Type Support	Lossless Compression Support
R8G8B8A8_UNORM	All	Read/Write	All
R8G8B8A8_UNORM_SRGB	All	Read/Write	All
R8G8B8A8_SNORM	All	Read/Write	All
R8G8B8A8_SINT	*Not Supported*	Read/Write	All
R8G8B8A8_UINT	*Not Supported*	Read/Write	All
R16G16_UNORM	All	Read/Write	All
R16G16_SNORM	All	Read/Write	All
R16G16_SINT	*Not Supported*	Read/Write	All
R16G16_UINT	*Not Supported*	Read/Write	All
R16G16_FLOAT	All	Read/Write	All
B10G10R10A2_UNORM	All	Read/Write	All
B10G10R10A2_UNORM_SRGB	All	Read/Write	All
R10G10B10_FLOAT_A2_UNORM	All	Read/Write	All
R11G11B10_FLOAT	All	Read/Write	All
R32_SINT	*Not Supported*	Read/Write	All
R32_UINT	*Not Supported*	Read/Write	All
R32_FLOAT	All	Read/Write	All
B8G8R8X8_UNORM	All	Read/Write	All
B8G8R8X8_UNORM_SRGB	All	Read/Write	All
B5G6R5_UNORM	All	Read/Write	All
B5G6R5_UNORM_SRGB	All	Read/Write	All
B5G5R5A1_UNORM	All	Read/Write	All
B5G5R5A1_UNORM_SRGB	All	Read/Write	All
B4G4R4A4_UNORM	All	Read/Write	All
B4G4R4A4_UNORM_SRGB	All	Read/Write	All
R8G8_UNORM	All	Read/Write	All
R8G8_SNORM	All	Read/Write	All
R8G8_SINT	*Not Supported*	Read/Write	All
R8G8_UINT	*Not Supported*	Read/Write	All
R16_UNORM	All	Read/Write	All
R16_SNORM	All	Read/Write	All
R16_SINT	*Not Supported*	Read/Write	All
R16_UINT	*Not Supported*	Read/Write	All
R16_FLOAT	All	Read/Write	All
B5G5R5X1_UNORM	All	Read/Write	All
B5G5R5X1_UNORM_SRGB	All	Read/Write	All
A1B5G5R5_UNORM	All	Read/Write	All

Surface Format Name	Color Blend Support	Message Type Support	Lossless Compression Support
A4B4G4R4_UNORM	All	Read/Write	All
R8_UNORM	All	Read/Write	All
R8_SNORM	All	Read/Write	All
R8_SINT	*Not Supported*	Read/Write	All
R8_UINT	*Not Supported*	Read/Write	All
A8_UNORM	All	Read/Write	All
YCRCB_NORMAL	*Not Supported*	Write	*Not Supported*
YCRCB_SWAPUVY	*Not Supported*	Write	*Not Supported*
YCRCB_SWAPUV	*Not Supported*	Write	*Not Supported*
YCRCB_SWAPY	*Not Supported*	Write	*Not Supported*

Subspan/Pixel to Slot Mapping

Pixels are numbered as follows within a subspan:

0 = upper left

1 = upper right

2 = lower left

3 = lower right

Programming Note	
Context:	Render Target Surfaces
• When SIMD32 or SIMD16 PS threads send render target writes with multiple SIMD8 and SIMD16 messages, the following must hold: • All the slots (as described above) must have a corresponding render target write irrespective of the slot's validity. A slot is considered valid when at least one sample is enabled. For example, a SIMD16 PS thread must send two SIMD8 render target writes to cover all the slots.	

Render Target Messages

The message operations on Render Cache Data Port use the same interface as the other data port messages. The message's Descriptor is sent to the target data port on a message sideband bus, and the other parts are sequentially transmitted between the data port and EU registers across the OBUS. The total number of registers sent and received by the data port is provided to the data port on the message sideband bus.

Descriptor	Describes the Message Type, the Message Specific Controls, and whether a Message Header is present. The Message Descriptor and the destination Data Port (SFID) are encoded as part of the *send* instruction.
Header	When present, provides additional Message specific controls. The message header is optional for some Render Target messages. When present, it is a 2-register payload.
Source Payload	Provides source data for write operations.
Writeback Payload	Returns result data for read operations.

The next sections describe all of the supported formats for the Source and Writeback Data Payloads, the Message Descriptors, and their Message Headers.

Message Summary tables describe the parameters and characteristics of the Render Target data port messages, following the notational conventions used in Message Formats.

Message-Specific Descriptors

The Render Cache data port has only 2 message types: Render Target Write and Render Target Read. The Render Target Write message type has 5 subtype operations. The Render Target Read message type has 2 subtype operations. The specific message descriptors for these messages are below.

Render Target Write Message

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	W	BTS	1D, 2D, 3D, CUBE, BUFFER	8, 16	1	(Surface[U, V, R, LOD])->SurfaceFormat[Slot]	Surface	RTPSM

This message writes 8 or 16 pixels to a render target. Depending on parameters contained in the message and state, it may also perform a depth and stencil buffer write and/or a render target read for a color blend operation. Additional operations enabled in the Color Calculator state are also initiated as a result of issuing this message (depth test, alpha test, logic ops, etc.). This message is intended only for use by pixel shader kernels for writing results to render targets.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	{Opt} MH_RT	{Forbidden}	MDP_RTW_8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	MH_RT_ZM	{Forbidden}	MDP_RTW_ZM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	MH_RT_Z	{Forbidden}	MDP_RTW_Z8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	MH_RT_M	{Forbidden}	MDP_RTW_M8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	{Opt} MH_RT	{Forbidden}	MDP_RTW_8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	MH_RT_ZM	{Forbidden}	MDP_RTW_ZM8DS	{Forbidden}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	MH_RT_Z	{Forbidden}	MDP_RTW_Z8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	MH_RT_M	{Forbidden}	MDP_RTW_M8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	{Opt} MH_RT	{Forbidden}	MDP_RTW_8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_ZMA	{Forbidden}	MDP_RTW_ZMA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_ZA	{Forbidden}	MDP_RTW_ZA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_MA	{Forbidden}	MDP_RTW_MA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_ZM	{Forbidden}	MDP_RTW_ZM8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_Z	{Forbidden}	MDP_RTW_Z8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_A	{Forbidden}	MDP_RTW_A8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_M	{Forbidden}	MDP_RTW_M8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	{Opt} MH_RT	{Forbidden}	MDP_RTW_S8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	MH_RT_ZM	{Forbidden}	MDP_RTW_SZM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	MH_RT_Z	{Forbidden}	MDP_RTW_SZ8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_LO8DS	MH_RT_M	{Forbidden}	MDP_RTW_SM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	{Opt} MH_RT	{Forbidden}	MDP_RTW_S8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	MH_RT_ZM	{Forbidden}	MDP_RTW_SZM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	MH_RT_Z	{Forbidden}	MDP_RTW_SZ8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_HI8DS	MH_RT_M	{Forbidden}	MDP_RTW_SM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	{Opt} MH_RT	{Forbidden}	MDP_RTW_S8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_ZMA	{Forbidden}	MDP_RTW_SZMA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_ZA	{Forbidden}	MDP_RTW_SZA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_MA	{Forbidden}	MDP_RTW_SMA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_ZM	{Forbidden}	MDP_RTW_SZM8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_Z	{Forbidden}	MDP_RTW_SZ8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_A	{Forbidden}	MDP_RTW_SA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTW_SIMD8	MH_RT_M	{Forbidden}	MDP_RTW_SM8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	{Opt} MH_RT	{Forbidden}	MDP_RTW_16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	MH_RT_ZMA	{Forbidden}	MDP_RTW_ZMA16	{Forbidden}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	MH_RT_ZA	{Forbidden}	MDP_RTW_ZA16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	MH_RT_MA	{Forbidden}	MDP_RTW_MA16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	MH_RT_ZM	{Forbidden}	MDP_RTW_ZM16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	MH_RT_Z	{Forbidden}	MDP_RTW_Z16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	MH_RT_A	{Forbidden}	MDP_RTW_A16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_SIMD16	MH_RT_M	{Forbidden}	MDP_RTW_M16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_REP16	{Opt} MH_RT	{Forbidden}	MDP_RTW_16REP	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTW_REP16	MH_RT_M	{Forbidden}	MDP_RTW_M16REP	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	{Opt} MH_RT	{Forbidden}	MDP_RTWH_8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	MH_RT_ZM	{Forbidden}	MDP_RTWH_ZM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	MH_RT_Z	{Forbidden}	MDP_RTWH_Z8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	MH_RT_M	{Forbidden}	MDP_RTWH_M8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	{Opt} MH_RT	{Forbidden}	MDP_RTWH_8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	MH_RT_ZM	{Forbidden}	MDP_RTWH_ZM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	MH_RT_Z	{Forbidden}	MDP_RTWH_Z8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	MH_RT_M	{Forbidden}	MDP_RTWH_M8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	{Opt} MH_RT	{Forbidden}	MDP_RTWH_8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_ZMA	{Forbidden}	MDP_RTWH_ZMA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_ZA	{Forbidden}	MDP_RTWH_ZA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_MA	{Forbidden}	MDP_RTWH_MA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_ZM	{Forbidden}	MDP_RTWH_ZM8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_Z	{Forbidden}	MDP_RTWH_Z8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_A	{Forbidden}	MDP_RTWH_A8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_M	{Forbidden}	MDP_RTWH_M8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	{Opt} MH_RT	{Forbidden}	MDP_RTWH_S8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	MH_RT_ZM	{Forbidden}	MDP_RTWH_SZM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	MH_RT_Z	{Forbidden}	MDP_RTWH_SZ8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_LO8DS	MH_RT_M	{Forbidden}	MDP_RTWH_SM8DS	{Forbidden}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	{Opt} MH_RT	{Forbidden}	MDP_RTWH_S8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	MH_RT_ZM	{Forbidden}	MDP_RTWH_SZM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	MH_RT_Z	{Forbidden}	MDP_RTWH_SZ8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_HI8DS	MH_RT_M	{Forbidden}	MDP_RTWH_SM8DS	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	{Opt} MH_RT	{Forbidden}	MDP_RTWH_S8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_ZMA	{Forbidden}	MDP_RTWH_SZMA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_ZA	{Forbidden}	MDP_RTWH_SZA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_MA	{Forbidden}	MDP_RTWH_SMA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_ZM	{Forbidden}	MDP_RTWH_SZM8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_Z	{Forbidden}	MDP_RTWH_SZ8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_A	{Forbidden}	MDP_RTWH_SA8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	1	MSD_RTWH_SIMD8	MH_RT_M	{Forbidden}	MDP_RTWH_SM8	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	{Opt} MH_RT	{Forbidden}	MDP_RTWH_16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_ZMA	{Forbidden}	MDP_RTWH_ZMA16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_ZA	{Forbidden}	MDP_RTWH_ZA16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_MA	{Forbidden}	MDP_RTWH_MA16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_ZM	{Forbidden}	MDP_RTWH_ZM16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_Z	{Forbidden}	MDP_RTWH_Z16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_A	{Forbidden}	MDP_RTWH_A16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_SIMD16	MH_RT_M	{Forbidden}	MDP_RTWH_M16	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_REP16	{Opt} MH_RT	{Forbidden}	MDP_RTWH_16REP	{Forbidden}
W	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	1	MSD_RTWH_REP16	MH_RT_M	{Forbidden}	MDP_RTWH_M16REP	{Forbidden}

The sample for a pixel is killed (not written to the render target or depth buffer) if the corresponding oMask bit is zero. Bits in oMask larger than the number of multisamples are ignored.

The color payload is included if the Message Type is SIMD8 single source or SIMD8 Image Write.

The SIMD16 Replicated Data Color payload is included if the Message Type specifies single source message with replicated data. One set of R/G/B/A data is included in the message, and this data is replicated to all 16 pixels. The Replicated SIMD16 message is legal with color data and oMask, but the registers for depth, stencil, and antialias alpha data cannot be included with this message, and the corresponding bits in the message header must indicate that these registers are not present.

The following table enumerates the order that the data payload is packed in registers: Source 0 Alpha (s0A), Output Mask (oM), Pixel components (R, G, B, A), Source Depth (sZ), and Output Stencil (oS).

Summary of Data Payload Register Order for Render Target Write Messages

Message Type	Present?				Message Register												
	S0A	oM	sZ	oS	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
SIMD16	0	0	0	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A					
SIMD16	1	0	0	0	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A			
SIMD16	0	0	1	0	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ			
SIMD16	1	0	1	0	1/0s0A	3/2s0A	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ	
SIMD16	0	1	0	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A				
SIMD16	1	1	0	0	1/0soA	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A		
SIMD16	0	1	1	0	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ		
SIMD16	1	1	1	0	1/0s0A	3/2s0A	oM	1/0R	3/2R	1/0G	3/2G	1/0B	3/2B	1/0A	3/2A	1/0sZ	3/2sZ
REP16	0	0	0	0	RGBA												
LO8DS	0	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
LO8DS	0	0	1	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
LO8DS	0	1	0	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A				
LO8DS	0	1	1	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ			
HI8DS	0	0	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A					
HI8DS	0	0	1	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ				
HI8DS	0	1	0	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A				
HI8DS	0	1	1	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ			
SIMD8	0	0	0	0	R	G	B	A									
SIMD8	1	0	0	0	s0A	R	G	B	A								
SIMD8	0	0	1	0	R	G	B	A	sZ								
SIMD8	1	0	1	0	s0A	R	G	B	A	sZ							
SIMD8	0	1	0	0	oM	R	G	B	A								
SIMD8	1	1	0	0	s0A	oM	R	G	B	A							
SIMD8	0	1	1	0	oM	R	G	B	A	sZ							
SIMD8	1	1	1	0	s0A	oM	R	G	B	A	sZ						
LO8DS	0	0	0	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS				
LO8DS	0	0	1	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0OoS			
LO8DS	0	1	0	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS			
LO8DS	0	1	1	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0oS		
HI8DS	0	0	0	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS				
HI8DS	0	0	1	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS			
HI8DS	0	1	0	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS			
HI8DS	0	1	1	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS		
SIMD8	0	0	0	1	R	G	B	A	oS								
SIMD8	1	0	0	1	s0A	R	G	B	A	oS							
SIMD8	0	0	1	1	R	G	B	A	sZ	oS							
SIMD8	1	0	1	1	s0A	R	G	B	A	sZ	oS						
SIMD8	0	1	0	1	oM	R	G	B	A	oS							
SIMD8	1	1	0	1	s0A	oM	R	G	B	A	oS						
SIMD8	0	1	1	1	oM	R	G	B	A	sZ	oS						
SIMD8	1	1	1	1	s0A	oM	R	G	B	A	sZ	oS					
HP SIMD16	0	0	0	0	R	G	B	A									

HP SIMD16	1	0	0	0	s0A	R	G	B	A								
HP SIMD16	0	0	1	0	R	G	B	A	sZ								
HP SIMD16	1	0	1	0	s0A	R	G	B	A	sZ							
HP SIMD16	0	1	0	0	oM	R	G	B	A								
HP SIMD16	1	1	0	0	s0A	oM	R	G	B	A							
HP SIMD16	0	1	1	0	oM	R	G	B	A	sZ							
HP SIMD16	1	1	1	0	s0A	oM	R	G	B	A	sZ						
HP REP16	0	0	0	0	RGBA												
HP LO8DS	0	0	0	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A					
HP LO8DS	0	0	1	0	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ				
HP LO8DS	0	1	0	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A				
HP LO8DS	0	1	1	0	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ			
HP HI8DS	0	0	0	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A					
HP HI8DS	0	0	1	0	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ				
HP HI8DS	0	1	0	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A				
HP HI8DS	0	1	1	0	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ			
HP SIMD8	0	0	0	0	R	G	B	A									
HP SIMD8	1	0	0	0	s0A	R	G	B	A								
HP SIMD8	0	0	1	0	R	G	B	A	sZ								
HP SIMD8	1	0	1	0	s0A	R	G	B	A	sZ							
HP SIMD8	0	1	0	0	oM	R	G	B	A								
HP SIMD8	1	1	0	0	s0A	oM	R	G	B	A							
HP SIMD8	0	1	1	0	oM	R	G	B	A	sZ							
HP SIMD8	1	1	1	0	s0A	oM	R	G	B	A	sZ						
HP LO8DS	0	0	0	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS				
HP LO8DS	0	0	1	1	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0OoS			
HP LO8DS	0	1	0	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0oS			
HP LO8DS	0	1	1	1	oM	1/0s0R	1/0s0G	1/0s0B	1/0s0A	1/0s1R	1/0s1G	1/0s1B	1/0s1A	1/0sZ	1/0oS		
HP HI8DS	0	0	0	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS				
HP HI8DS	0	0	1	1	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS			
HP HI8DS	0	1	0	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2oS			
HP HI8DS	0	1	1	1	oM	3/2s0R	3/2s0G	3/2s0B	3/2s0A	3/2s1R	3/2s1G	3/2s1B	3/2s1A	3/2sZ	3/2oS		
HP SIMD8	0	0	0	1	R	G	B	A	oS								
HP SIMD8	1	0	0	1	s0A	R	G	B	A	oS							
HP SIMD8	0	0	1	1	R	G	B	A	sZ	oS							
HP SIMD8	1	0	1	1	s0A	R	G	B	A	sZ	oS						
HP SIMD8	0	1	0	1	oM	R	G	B	A	oS							
HP SIMD8	1	1	0	1	s0A	oM	R	G	B	A	oS						
HP SIMD8	0	1	1	1	oM	R	G	B	A	sZ	oS						
HP SIMD8	1	1	1	1	s0A	oM	R	G	B	A	sZ	oS					

Programming Note	
Context:	Render Target Write Message
Typically, the last message in a pixel shader is a Render Target Write message, with EOT set.	

Programming Note	
Context:	Render Target Write Message
• The dual source message cannot be used if the Render Target Rotation field in SURFACE_STATE is set to anything other than RTROTATE_0DEG. • If multiple SIMD8 Dual Source messages are delivered by the pixel shader thread, each SIMD8_DUALSRC_LO message must be issued *before* the SIMD8_DUALSRC_HI message with the same Slot Group Select setting. • Output Stencil is not supported with SIMD16 Render Target Write Messages.	

Programming Note	
Context:	Render Target Writes in Multirate Shaders
• In multirate shaders, render target writes that can potentially kill pixels (via oMask, oDepth, or oStencil) cannot modify the oMask, oDepth, or oStencil value after writing to a render target. For instance, if a pixel-rate render target is written, an associated sample-rate render target cannot modify these values or undefined behavior can occur. • In multirate shaders that write to render targets in multiple stages, render target 0 must be written to before any other render targets. • Pixel shaders dispatched at the coarse rate that contain a SIMD16 pixel-rate shader, cannot use SIMD8 messages, preventing a SIMD16 pixel-rate (or sample-rate) shader from using Dual Source messages.	

Replicate Data

The replicate data or replicate color render target message (REP16) is used for clearing Render Target. This message performs better than the other messages due to its smaller message length. This message does not support depth, stencil, or antialias alpha data being sent with it.

The pixel scoreboard bits corresponding to the dispatched pixel mask are cleared only if the Last Render Target Select bit is set in the message descriptor.

Programming Note	
Context:	Replicate Data Render Target Write Message
Replicate Data Render Target Write message is not supported with AA alpha i.e., when primitive has Anti-Aliased alpha enabled e.g., AA lines and points.	

Programming Note	
Context:	Replicate Data Render Target Write Message
Replicate Data Render Target Write message should not be used on all projects.	

Single Source

The "normal" render target messages are single source. There are two forms, SIMD16 and SIMD8, intended for the equivalent-sized pixel shader threads. A single source (4 channels) is delivered for each of the 16 or 8 pixels in the message payload. Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

The pixel scoreboard bits corresponding to the dispatched pixel mask (or half of the mask in the case of SIMD8 messages) are cleared only if the Last Render Target Select bit is set in the message descriptor. However, if Last Render Target Select is set, the message still causes pixel scoreboard clear and depth/stencil buffer updates if enabled.

Dual Source

The dual source render target messages only have SIMD8 forms due to maximum message length limitations. SIMD16 pixel shaders must send two of these messages to cover all of the pixels. Each message contains two colors (4 channels each) for each pixel in the message payload. In addition to the first source, the second source can be selected as a blend factor (BLENDFACTOR_*_SRC1_* options in the blend factor fields of COLOR_CALC_STATE or BLEND_STATE). Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

For SIMD16 PS thread with two output colors with SRC1 as one of the blend factors with blending enabled, SW must send messages in the following sequence for each RT: SIMD8 dual source (SRC0 and SRC1) RTW message (low); SIMD8 dual source RTW message (high); SIMD16 single src (SRC1) RTW message with second color. If blending is not enabled OR if blending is enabled without SRC1 as a blend factor, SW continues to send regular SIMD16 SRC0 and SRC1 messages.

CPS H/W does not support SIMD8+LRTS messages for a Fused SIMD32 dispatch and results in improper deallocation/ dereferences to pointers and entries in the Pixel Scoreboard and local PQ buffers. Since dual source render target messages can only have SIMD8 forms due to maximum message length limitations, a SIMD8 message with LRTS from a SIMD32 kernel is incompatible with the hardware in place.

Render Target Read Message

Addr Align	Data Width	R/W	Address Model	Surface Type	SIMD Slots	Data Elements	SIMD Address Calculation	Bounds Check	Execution Mask
DW	DW	R	BTS	1D, 2D, 3D, CUBE, BUFFER	8, 16	4	(Surface[U, V, R, LOD])->DW[Chan]	Surface	RTPSM

This message takes 8 or 16 pixels for reads to a render target. This message is intended only for use by pixel shader kernels for reading data from render targets.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD8	{4}	MSD_RTR_SIMD8	MH_RT	{Forbidden}	{Forbidden}	{4} MDP_DW_SIMD16
R	BTS	1D, 2D, 3D, CUBE, BUFFER	SIMD16	{4}	MSD_RTR_SIMD16	MH_RT	{Forbidden}	{Forbidden}	{4} MDP_DW_SIMD8

A SIMD8 writeback message consists of 4 destination registers. A SIMD16 writeback message consists of 8 destination registers.

Restrictions:

Must not have End-of-Thread bit set in message.

This message can only be issued from a kernel specified in WM_STATE or 3DSTATE_WM (pixel shader kernel), dispatched in non-contiguous mode. Any other kernel issuing this message causes undefined behavior.

Render Target read on an MSRT is supported via per-sample pixel shader.

Message Data Payloads

The data payloads for Render Target Read messages depend on the SIMD message subtype and the channel enables in the message descriptor. The writeback data payloads are 4 channels of red, green, blue, and alpha data. Each channel is one or two message registers laid out in the standard 8 or 16 DWord format, depending on whether it is a SIMD8 or SIMD16 data operation.

The data payloads for Render Target Write messages depend on the Message Subtype from the message descriptor, and the settings of the oMask, Source 0 Alpha, and Source Depth presence control bits in the message header. The following data tables list all the valid source payloads combinations for the Render Target Write message.

The half precision Render Target Write messages have data payloads that can pack a full SIMD16 payload into 1 register instead of two. The half-precision packed format is used for RGBA and Source 0 Alpha, but Source Depth data payload is always supplied in full precision.

Render Target Data Payloads

This section contains various registers for the Render Target Data Payloads.

MDP_RTW_16 - SIMD16 Render Target Data Payload

MDP_RTW_ZMA16 - SZ OM S0A SIMD16 Render Target Data Payload

MDP_RTW_ZA16 - SZ S0A SIMD16 Render Target Data Payload

MDP_RTW_MA16 - OM S0A SIMD16 Render Target Data Payload

MDP_RTW_ZM16 - SZ OM SIMD16 Render Target Data Payload

MDP_RTW_Z16 - SZ SIMD16 Render Target Data Payload

MDP_RTW_A16 - S0A SIMD16 Render Target Data Payload

MDP_RTW_M16 - OM SIMD16 Render Target Data Payload

MDP_RTW_16REP - Replicated SIMD16 Render Target Data Payload

MDP_RTW_M16REP - OM Replicated SIMD16 Render Target Data Payload

MDP_RTW_8 - SIMD8 Render Target Data Payload

MDP_RTW_ZMA8 - SZ OM S0A SIMD8 Render Target Data Payload

MDP_RTW_ZM8 - SZ OM SIMD8 Render Target Data Payload

MDP_RTW_ZA8 - SZ S0A SIMD8 Render Target Data Payload

MDP_RTW_MA8 - OM S0A SIMD8 Render Target Data Payload

MDP_RTW_Z8 - SZ SIMD8 Render Target Data Payload

MDP_RTW_A8 - S0A SIMD8 Render Target Data Payload

MDP_RTW_M8 - OM SIMD8 Render Target Data Payload

MDP_RTW_S8 - OS SIMD8 Render Target Data Payload

MDP_RTW_SZMA8 - OS SZ OM S0A SIMD8 Render Target Data Payload

MDP_RTW_SZM8 - OS SZ OM SIMD8 Render Target Data Payload

MDP_RTW_SZA8 - OS SZ S0A SIMD8 Render Target Data Payload

MDP_RTW_SMA8 - OS OM S0A SIMD8 Render Target Data Payload

MDP_RTW_SZ8 - OS SZ SIMD8 Render Target Data Payload

MDP_RTW_SA8 - OS S0A SIMD8 Render Target Data Payload

MDP_RTW_SM8 - OS OM SIMD8 Render Target Data Payload

MDP_RTW_8DS - SIMD8 Dual Source Render Target Data Payload

MDP_RTW_8DS - SIMD8 Dual Source Render Target Data Payload

MDP_RTW_ZM8DS - SZ OM SIMD8 Dual Source Render Target Data Payload

MDP_RTW_Z8DS - SZ SIMD8 Dual Source Render Target Data Payload

MDP_RTW_M8DS - OM SIMD8 Dual Source Render Target Data Payload

MDP_RTW_S8DS - OS SIMD8 Dual Source Render Target Data Payload

MDP_RTW_M8DS - OM SIMD8 Dual Source Render Target Data Payload

MDP_RTW_SZM8DS - OS SZ OM SIMD8 Dual Source Render Target Data Payload

MDP_RTW_SZ8DS - OS SZ SIMD8 Dual Source Render Target Data Payload

MDP_RTWH_16 - Half Precision SIMD16 Render Target Data Payload

MDP_RTW_SZ8DS - OS SZ SIMD8 Dual Source Render Target Data Payload

MDP_RTWH_ZM16 - Half Precision SZ OM SIMD16 Render Target Data Payload

MDP_RTWH_ZMA16 - Half Precision SZ OM S0A SIMD16 Render Target Data Payload

MDP_RTWH_ZA16 - Half Precision SZ S0A SIMD16 Render Target Data Payload

MDP_RTWH_MA16 - Half Precision OM S0A SIMD16 Render Target Data Payload

MDP_RTWH_Z16 - Half Precision SZ SIMD16 Render Target Data Payload

MDP_RTWH_A16 - Half Precision S0A SIMD16 Render Target Data Payload

MDP_RTWH_M16 - Half Precision OM SIMD16 Render Target Data Payload

MDP_RTWH_16REP - Half Precision Replicated SIMD16 Render Target Data Payload

MDP_RTWH_M16REP - Half Precision OM Replicated SIMD16 Render Target Data Payload

MDP_RTWH_8 - Half Precision SIMD8 Render Target Data Payload

MDP_RTWH_ZMA8 - Half Precision SZ OM S0A SIMD8 Render Target Data Payload

MDP_RTWH_ZM8 - Half Precision SZ OM SIMD8 Render Target Data Payload

MDP_RTWH_ZA8 - Half Precision SZ S0A SIMD8 Render Target Data Payload

MDP_RTWH_MA8 - Half Precision OM S0A SIMD8 Render Target Data Payload

MDP_RTWH_Z8 - Half Precision SZ SIMD8 Render Target Data Payload

MDP_RTWH_A8 - Half Precision S0A SIMD8 Render Target Data Payload

MDP_RTWH_M8 - Half Precision OM SIMD8 Render Target Data Payload

MDP_RTWH_S8 - Half Precision OS SIMD8 Render Target Data Payload

MDP_RTWH_SZMA8 - Half Precision OS SZ OM S0A SIMD8 Render Target Data Payload

MDP_RTWH_SZM8 - Half Precision OS SZ OM SIMD8 Render Target Data Payload

MDP_RTWH_SZA8 - Half Precision OS SZ S0A SIMD8 Render Target Data Payload

MDP_RTWH_SMA8 - Half Precision OS OM S0A SIMD8 Render Target Data Payload

MDP_RTWH_SZ8 - Half Precision OS SZ SIMD8 Render Target Data Payload

MDP_RTWH_SA8 - Half Precision OS S0A SIMD8 Render Target Data Payload

MDP_RTWH_SM8 - Half Precision OS OM SIMD8 Render Target Data Payload

MDP_RTWH_8DS - Half Precision SIMD8 Dual Source Render Target Data Payload

MDP_RTWH_ZM8 - Half Precision SZ OM SIMD8 Render Target Data Payload

MDP_RTWH_M8DS - Half Precision OM SIMD8 Dual Source Render Target Data Payload

MDP_RTWH_S8DS - Half Precision OS SIMD8 Dual Source Render Target Data Payload

MDP_RTWH_SZM8DS - Half Precision OS SZ OM SIMD8 Dual Source Render Target Data Payload

MDP_RTWH_SZ8DS - Half Precision OS SZ SIMD8 Dual Source Render Target Data Payload

MDP_RTWH_SM8DS - Half Precision OS OM SIMD8 Dual Source Render Target Data Payload

Shared Functions Pixel Interpolater

The Pixel Interpolator provides barycentric parameters at various offsets relative to the pixel location. These barycentric parameters are in the same format and layout as those received in the pixel shader dispatch. Please refer to the "Windower" chapter in the "3D Pipeline" volume for more details on barycentric parameters.

Barycentric parameters delivered in the pixel shader payload are at pre-defined positions based on **Barycentric Interpolation Mode** bits selected in 3DSTATE_WM. The pixel interpolator allows barycentric parameters to be computed at additional locations.

Messages

The following is the message definition for the Pixel Interpolator shared function.

Programming Note	
Context:	Messages
Pixel Interpolator messages can only be delivered by pixel shader kernels.	

Execution Mask. Each bit in the execution mask enables the corresponding slot's barycentric parameter return to the destination registers.

Initiating Message

Message Descriptor

Bits	Description
19	**Header Present:** Specifies whether the message includes a header phase. Must be zero for all *Pixel Interpolator* messages. Format = Enable
18:17	Ignored
16	**SIMD Mode.** Specifies the SIMD mode of the message being sent. Format = U1 0: SIMD8 mode 1: SIMD16 mode
15	**Shading Rate for Attribute Evaluation** This field indicates how to interpret the message type (bits 13:12). Format: U1 0: Evaluate at pixel shading rate. 1: Evaluate at coarse pixel shading rate.
14	**Interpolation Mode.** Specifies which interpolation mode is used. Format = U1 0: Perspective Interpolation 1: Linear Interpolation

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Programming Note</th></tr>
<tr><td>This field cannot be set to "Linear Interpolation" unless Non-Perspective Barycentric Enable in 3DSTATE_CLIP is enabled.</td></tr>
</table>
<table>
<tr><th colspan="2">Programming Note</th></tr>
<tr><td>Context:</td><td>Message Descriptor</td></tr>
<tr><td colspan="2">This field is ignored when Message Type is set to Coarse To Pixel Mapping message.</td></tr>
</table>
</td></tr>
<tr><td>13:12</td><td>
Message Type. Specifies the type of message being sent when pixel-rate evaluation is requested.
Format = U2
0: Per Message Offset (eval_snapped with immediate offset)
1: Sample Position Offset (eval_sindex)
2: Centroid Position Offset (eval_centroid)
3: Per Slot Offset (eval_snapped with register offset)
Message Type. Specifies the type of message being sent when coarse-rate evaluation is requested.
Format = U2
0: Coarse to Pixel Mapping Message (internal message)
1: Reserved
2: Coarse Centroid Position (eval_centroid)
3: Per Slot Coarse Pixel Offset (eval_snapped with register offset)
<table>
<tr><th colspan="2">Programming Note</th></tr>
<tr><td>Context:</td><td>Message Descriptor</td></tr>
<tr><td colspan="2">When Message Type is Sample Position, requesting an attribute at sample index beyond the range defined by the Forced Sample Count is illegal.</td></tr>
</table>
</td></tr>
<tr><td>11</td><td>
Note: For shaders dispatched at coarse rate, this bit has a different meaning and is part of the Pixel Shader Phase field.
Slot Group Select. This field selects whether slots 15:0 or slots 31:16 are used for bypassed data.
Bypassed data includes the X/Y addresses and centroid position. For 8- and 16-pixel dispatches, SLOTGRP_LO must be selected on every message. For 32-pixel dispatches, this field must be set correctly for each message based on which slots are currently being processed.
0: SLOTGRP_LO: Choose bypassed data for slots 15:0.
1: SLOTGRP_HI: Choose bypassed data for slots 31:16.
</td></tr>
</table>

Bits	Description
	Programming Note **Context:** Message Descriptor This field must be set to SLOTGRP_LO for SIMD8 messages. SIMD8 messages always use bypassed data for slots 7:0.
11:8	For shaders not dispatched at coarse rate, bit 11 is the **Slot Group Select** bit and bits 10:8 are ignored. **Pixel Shader Phase.** For shaders dispatched at coarse pixel rate, specifies the counter value of the inner PS loop inside monolithic CPS+PS(+S) shader code. The Pixel Interpolator uses this counter value to identify affected pixels for all attributes interpolated at pixel rate and to identify pixels requested for Coarse to Pixel Mapping messages. Format: U4 Range = [0..15] Values other than 0 are valid only for shaders dispatched at coarse shading rate.
7:0	**Message Specific Control.** Refer to the sections below for the definition of these bits based on **Message Type**.

"Per Message Offset" Message Descriptor

Bit	Description
7:4	**Per Message Y Pixel Offset** Specifies the Y Pixel Offset that applies to all slots. Format = S4 2's complement representing units of 1/16 pixel. Range = [-8/16, +7/16]
3:0	**Per Message X Pixel Offset** Specifies the X Pixel Offset that applies to all slots. Format = S4 2's complement representing units of 1/16 pixel. Range = [-8/16, +7/16]

"Sample Position Offset" Message Descriptor

Bits	Description
7:4	**Sample Index** Specifies the sample index that applies to all slots. Sample Index must not exceed the value of NUM_RASTSAMPLES when NUM_RASTSAMPLES > 1. From API, perspective, Forced Sample Count Defines the maximum allowable index in this message.

Bits	Description
	Format = U4 **Range** [0, 15]
3:0	Ignored

"Centroid Position" and "Per Slot Offset" Message Descriptor

Bit	Description
7:0	Ignored

"Coarse to Pixel Mapping" Message Descriptor

Bits	Description
7:0	**Coarse to Pixel Mapping Phase Enable bits** The Coarse to Pixel mapping message may include Output Pixel Coverage in the message payload. This payload is optional and delivered only when coarse phase of pixel shader computes output coverage mask or discards coarse pixels (output pixel coverage mask is different than input pixel coverage mask). The send instruction requires payload to consist of at least one GRF register. The following bit indicates if payload includes valid Output Pixel Coverage mask, or it is a dummy register to satisfy non-zero payload requirement. bit 5: Output Pixel Coverage Mask Valid in payload The Coarse to Pixel mapping message always returns two phases (two GRF registers) but it can also return optional phases, as enabled by the following bits: bit 4: Perspective Pixel Center Location Barycentric bit 3: Perspective Pixel Centroid Location Barycentric bit 2: Linear Pixel Center Location Barycentric bit 1: Linear Pixel Centroid Location Barycentric bit 0: Pixel Input Coverage Mask

Programming Note	
Context:	**Coarse to Pixel Mapping Message Descriptor**
Coarse pixel to pixel mapping message does not support Output Coverage Masks i.e. in the bit field "Coarse to Pixel Mapping Phase Enable bits" bit5 must be reset.	

Message Payload for Most Messages

This message payload applies to the following message types:

- Per Message Offset
- Sample Position Offset
- Centroid Position Offset

DWord	Bit	Description
M0.7:0		Ignored

SIMD8 Per Slot Offset Message Payload

This message payload applies only to the SIMD8 Per Slot Offset message type. The message length is 2.

DWord	Bit	Description
M0.7	31:0	**Slot 7 X Pixel Offset** Specifies the X pixel offset for slot 7. When CPsize > (1,1), interpretation of this field is wrt CPcenter with the following offset encodings as lsbs of the 32b offset.

Coarse pixel size	Indexable range	Representable range size	Number of bits needed {x, y}	Binary mask of usable bits
1x1 (fine)	{[-8, 7], [-8, 7]}	{16, 16}	{4, 4}	{000000000000xxxx, 000000000000xxxx}
1x2	{[-8, 7], [-16, 15]}	{16, 32}	{4, 5}	{000000000000xxxx, 00000000000xxxxx}
2x1	{[-16, 15], [-8, 7]}	{32, 16}	{5, 4}	{00000000000xxxxx, 000000000000xxxx}
2x2	{[-16, 15], [-16, 15]}	{32, 32}	{5, 5}	{00000000000xxxxx, 00000000000xxxxx}
2x4	{[-16, 15], [-32, 31]}	{32, 64}	{5, 6}	{00000000000xxxxx, 0000000000xxxxxx}
4x2	{[-32, 31], [-16, 15]}	{64, 32}	{6, 5}	{0000000000xxxxxx, 00000000000xxxxx}
4x4	{[-32, 31], [-32, 31]}	{64, 64}	{6, 6}	{0000000000xxxxxx, 0000000000xxxxxx}

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td></td><td></td><td>
The tables below are a guide for conversion to from the fixed-point to decimal and fractional representation. The first usable bit in the binary mask is the sign bit, and the rest of the binary mask comprise the numerical portion.

The number scheme for four-bit values passed into EvaluateAttributeSnapped is not anything new brought about by variable rate shading. It is reiterated here for completeness.

For four-bit values:
<table>
<tr><th>Binary value</th><th>Decimal</th><th>Fractional</th></tr>
<tr><td>1000</td><td>-0.5f</td><td>-8 / 16</td></tr>
<tr><td>1001</td><td>-0.4375f</td><td>-7 / 16</td></tr>
<tr><td>1010</td><td>-0.375f</td><td>-6 / 16</td></tr>
<tr><td>1011</td><td>-0.3125f</td><td>-5 / 16</td></tr>
<tr><td>1100</td><td>-0.25f</td><td>-4 / 16</td></tr>
<tr><td>1101</td><td>-0.1875f</td><td>-3 / 16</td></tr>
<tr><td>1110</td><td>-0.125f</td><td>-2 / 16</td></tr>
<tr><td>1111</td><td>-0.0625f</td><td>-1 /16</td></tr>
<tr><td>0000</td><td>0.0f</td><td>0 / 16</td></tr>
<tr><td>0001</td><td>-0.0625f</td><td>1 / 16</td></tr>
<tr><td>0010</td><td>-0.125f</td><td>2 / 16</td></tr>
<tr><td>0011</td><td>-0.1875f</td><td>3 / 16</td></tr>
<tr><td>0100</td><td>-0.25f</td><td>4 / 16</td></tr>
<tr><td>0101</td><td>-0.3125f</td><td>5 / 16</td></tr>
<tr><td>0110</td><td>-0.375f</td><td>6 / 16</td></tr>
<tr><td>0111</td><td>-0.4375f</td><td>7 / 16</td></tr>
</table>
For five-bit values:
</td></tr>
</table>

DWord	Bit	Description

Binary value	Decimal	Fractional
10000	-1	-16 / 16
10001	-0.9375	-15 / 16
10010	-0.875	-14 / 16
10011	-0.8125	-13 / 16
10100	-0.75	-12 / 16
10101	-0.6875	-11 / 16
10110	-0.625	-10 / 16
10111	-0.5625	-9 / 16
11000	-0.5	-8 / 16
11001	-0.4375	-7 / 16
11010	-0.375	-6 / 16
11011	-0.3125	-5 / 16
11100	-0.25	-4 / 16
11101	-0.1875	-3 / 16
11110	-0.125	-2 / 16
11111	-0.0625	-1 / 16
00000	0	0 / 16
00001	0.0625	1 / 16
00010	0.125	2 / 16
00011	0.1875	3 / 16
00100	0.25	4 / 16

DWord	Bit	Description

00101	0.3125	5 / 16
00110	0.375	6 / 16
00111	0.4375	7 / 16
01000	0.5	8 / 16
01001	0.5625	9 / 16
01010	0.625	10 / 16
01011	0.6875	11 / 16
01100	0.75	12 / 16
01101	0.8125	13 / 16
01110	0.875	14 / 16
01111	0.9375	15 / 16

For six-bit values:

Binary value	Decimal	Fractional
100000	-2	-32 / 16
100001	-1.9375	-31 / 16
100010	-1.875	-30 / 16
100011	-1.8125	-29 / 16
100100	-1.75	-28 / 16
100101	-1.6875	-27 / 16
100110	-1.625	-26 / 16
100111	-1.5625	-25 / 16
101000	-1.5	-24 / 16

DWord	Bit	Description

101001	-1.4375	-23 / 16
101010	-1.375	-22 / 16
101011	-1.3125	-21 / 16
101100	-1.25	-20 / 16
101101	-1.1875	-19 / 16
101110	-1.125	-18 / 16
101111	-1.0625	-17 / 16
110000	-1	-16 / 16
110001	-0.9375	-15 / 16
110010	-0.875	-14 / 16
110011	-0.8125	-13 / 16
110100	-0.75	-12 / 16
110101	-0.6875	-11 / 16
110110	-0.625	-10 / 16
110111	-0.5625	-9 / 16
111000	-0.5	-8 / 16
111001	-0.4375	-7 / 16
111010	-0.375	-6 / 16
111011	-0.3125	-5 / 16
111100	-0.25	-4 / 16
111101	-0.1875	-3 / 16
111110	-0.125	-2 / 16

DWord	Bit	Description
		111111 -0.0625 -1 / 16 000000 0 0 / 16 000001 0.0625 1 / 16 000010 0.125 2 / 16 000011 0.1875 3 / 16 000100 0.25 4 / 16 000101 0.3125 5 / 16 000110 0.375 6 / 16 000111 0.4375 7 / 16 001000 0.5 8 / 16 001001 0.5625 9 / 16 001010 0.625 10 / 16 001011 0.6875 11 / 16 001100 0.75 12 / 16 001101 0.8125 13 / 16 001110 0.875 14 / 16 001111 0.9375 15 / 16 010000 1 16 / 16 010001 1.0625 17 / 16 010010 1.125 18 / 16 010011 1.1875 19 / 16 010100 1.25 20 / 16

DWord	Bit	Description
		<table><tr><td>010101</td><td>1.3125</td><td>21 / 16</td></tr><tr><td>010110</td><td>1.375</td><td>22 / 16</td></tr><tr><td>010111</td><td>1.4375</td><td>23 / 16</td></tr><tr><td>011000</td><td>1.5</td><td>24 / 16</td></tr><tr><td>011001</td><td>1.5625</td><td>25 / 16</td></tr><tr><td>011010</td><td>1.625</td><td>26 / 16</td></tr><tr><td>011011</td><td>1.6875</td><td>27 / 16</td></tr><tr><td>011100</td><td>1.75</td><td>28 / 16</td></tr><tr><td>011101</td><td>1.8125</td><td>29 / 16</td></tr><tr><td>011110</td><td>1.875</td><td>30 / 16</td></tr><tr><td>011111</td><td>1.9375</td><td>31 / 16</td></tr></table>
M0.6	31:0	**Slot 6 X Pixel Offset**
M0.5	31:0	**Slot 5 X Pixel Offset**
M0.4	31:0	**Slot 4 X Pixel Offset**
M0.3	31:0	**Slot 3 X Pixel Offset**
M0.2	31:0	**Slot 2 X Pixel Offset**
M0.1	31:0	**Slot 1 X Pixel Offset**
M0.0	31:0	**Slot 0 X Pixel Offset**
M1.7	31:0	**Slot 7 Y Pixel Offset** Specifies the Y pixel offset for slot 7. See **Slot 7 X Pixel Offset** for format
M1.6	31:0	**Slot 6 Y Pixel Offset**

DWord	Bit	Description
M1.5	31:0	**Slot 5 Y Pixel Offset**
M1.4	31:0	**Slot 4 Y Pixel Offset**
M1.3	31:0	**Slot 3 Y Pixel Offset**
M1.2	31:0	**Slot 2 Y Pixel Offset**
M1.1	31:0	**Slot 1 Y Pixel Offset**
M1.0	31:0	**Slot 0 Y Pixel Offset**

SIMD16 Per Slot Offset Message Payload

This message payload applies only to the SIMD16 Per Slot Offset message type. The message length is 4.

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td>M0.7</td><td>31:0</td><td>
Slot 7 X Pixel Offset
Specifies the X pixel offset for slot 7.
Format = S4 2's complement representing units of 1/16 pixel. The upper 28 bits are ignored.
Range = [-8/16, +7/16]
Slot 7 X Pixel Offset
Specifies the X pixel offset for slot 7.
Format = S4 2's complement representing units of 1/16 pixel. The upper 28 bits are ignored.
Range = [-8/16, +7/16]
When CPsize > (1,1), interpretation of this field is wrt CPcenter with the following offset encodings as lsbs of the 32b offset.
<table>
<tr><th>Coarse pixel size</th><th>Indexable range</th><th>Representable range size</th><th>Number of bits needed {x, y}</th><th>Binary mask of usable bits</th></tr>
<tr><td>1x1 (fine)</td><td>{[-8, 7], [-8, 7]}</td><td>{16, 16}</td><td>{4, 4}</td><td>{000000000000xxxx, 000000000000xxxx}</td></tr>
<tr><td>1x2</td><td>{[-8, 7], [-16, 15]}</td><td>{16, 32}</td><td>{4, 5}</td><td>{000000000000xxxx, 00000000000xxxxx}</td></tr>
<tr><td>2x1</td><td>{[-16, 15], [-8, 7]}</td><td>{32, 16}</td><td>{5, 4}</td><td>{00000000000xxxxx, 000000000000xxxx}</td></tr>
<tr><td>2x2</td><td>{[-16, 15], [-16,</td><td>{32, 32}</td><td>{5, 5}</td><td>{00000000000xxxxx,</td></tr>
</table>
</td></tr>
</table>

DWord	Bit	Description

	15]}			00000000000xxxxx}
2x4	{[-16, 15], [-32, 31]}	{32, 64}	{5, 6}	{00000000000xxxxx, 0000000000xxxxxx}
4x2	{[-32, 31], [-16, 15]}	{64, 32}	{6, 5}	{0000000000xxxxxx, 00000000000xxxxx}
4x4	{[-32, 31], [-32, 31]}	{64, 64}	{6, 6}	{0000000000xxxxxx, 0000000000xxxxxx}

The tables below are a guide for conversion to from the fixed-point to decimal and fractional representation. The first usable bit in the binary mask is the sign bit, and the rest of the binary mask comprise the numerical portion.

The number scheme for four-bit values passed into EvaluateAttributeSnapped is not anything new brought about by variable rate shading. It is reiterated here for completeness.

For four-bit values:

Binary value	Decimal	Fractional
1000	-0.5f	-8 / 16
1001	-0.4375f	-7 / 16
1010	-0.375f	-6 / 16
1011	-0.3125f	-5 / 16
1100	-0.25f	-4 / 16
1101	-0.1875f	-3 / 16
1110	-0.125f	-2 / 16
1111	-0.0625f	-1 /16
0000	0.0f	0 / 16
0001	-0.0625f	1 / 16
0010	-0.125f	2 / 16
0011	-0.1875f	3 / 16

DWord	Bit	Description

0100	-0.25f	4 / 16
0101	-0.3125f	5 / 16
0110	-0.375f	6 / 16
0111	-0.4375f	7 / 16

For five-bit values:

Binary value	Decimal	Fractional
10000	-1	-16 / 16
10001	-0.9375	-15 / 16
10010	-0.875	-14 / 16
10011	-0.8125	-13 / 16
10100	-0.75	-12 / 16
10101	-0.6875	-11 / 16
10110	-0.625	-10 / 16
10111	-0.5625	-9 / 16
11000	-0.5	-8 / 16
11001	-0.4375	-7 / 16
11010	-0.375	-6 / 16
11011	-0.3125	-5 / 16
11100	-0.25	-4 / 16
11101	-0.1875	-3 / 16
11110	-0.125	-2 / 16
11111	-0.0625	-1 / 16

DWord	Bit	Description

00000	0	0 / 16
00001	0.0625	1 / 16
00010	0.125	2 / 16
00011	0.1875	3 / 16
00100	0.25	4 / 16
00101	0.3125	5 / 16
00110	0.375	6 / 16
00111	0.4375	7 / 16
01000	0.5	8 / 16
01001	0.5625	9 / 16
01010	0.625	10 / 16
01011	0.6875	11 / 16
01100	0.75	12 / 16
01101	0.8125	13 / 16
01110	0.875	14 / 16
01111	0.9375	15 / 16

For six-bit values:

Binary value	Decimal	Fractional
100000	-2	-32 / 16
100001	-1.9375	-31 / 16
100010	-1.875	-30 / 16
100011	-1.8125	-29 / 16

DWord	Bit	Description

100100	-1.75	-28 / 16
100101	-1.6875	-27 / 16
100110	-1.625	-26 / 16
100111	-1.5625	-25 / 16
101000	-1.5	-24 / 16
101001	-1.4375	-23 / 16
101010	-1.375	-22 / 16
101011	-1.3125	-21 / 16
101100	-1.25	-20 / 16
101101	-1.1875	-19 / 16
101110	-1.125	-18 / 16
101111	-1.0625	-17 / 16
110000	-1	-16 / 16
110001	-0.9375	-15 / 16
110010	-0.875	-14 / 16
110011	-0.8125	-13 / 16
110100	-0.75	-12 / 16
110101	-0.6875	-11 / 16
110110	-0.625	-10 / 16
110111	-0.5625	-9 / 16
111000	-0.5	-8 / 16
111001	-0.4375	-7 / 16

DWord	Bit	Description

111010	-0.375	-6 / 16
111011	-0.3125	-5 / 16
111100	-0.25	-4 / 16
111101	-0.1875	-3 / 16
111110	-0.125	-2 / 16
111111	-0.0625	-1 / 16
000000	0	0 / 16
000001	0.0625	1 / 16
000010	0.125	2 / 16
000011	0.1875	3 / 16
000100	0.25	4 / 16
000101	0.3125	5 / 16
000110	0.375	6 / 16
000111	0.4375	7 / 16
001000	0.5	8 / 16
001001	0.5625	9 / 16
001010	0.625	10 / 16
001011	0.6875	11 / 16
001100	0.75	12 / 16
001101	0.8125	13 / 16
001110	0.875	14 / 16
001111	0.9375	15 / 16

<table>
<tr><th>DWord</th><th>Bit</th><th>Description</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><td>010000</td><td>1</td><td>16 / 16</td></tr>
<tr><td>010001</td><td>1.0625</td><td>17 / 16</td></tr>
<tr><td>010010</td><td>1.125</td><td>18 / 16</td></tr>
<tr><td>010011</td><td>1.1875</td><td>19 / 16</td></tr>
<tr><td>010100</td><td>1.25</td><td>20 / 16</td></tr>
<tr><td>010101</td><td>1.3125</td><td>21 / 16</td></tr>
<tr><td>010110</td><td>1.375</td><td>22 / 16</td></tr>
<tr><td>010111</td><td>1.4375</td><td>23 / 16</td></tr>
<tr><td>011000</td><td>1.5</td><td>24 / 16</td></tr>
<tr><td>011001</td><td>1.5625</td><td>25 / 16</td></tr>
<tr><td>011010</td><td>1.625</td><td>26 / 16</td></tr>
<tr><td>011011</td><td>1.6875</td><td>27 / 16</td></tr>
<tr><td>011100</td><td>1.75</td><td>28 / 16</td></tr>
<tr><td>011101</td><td>1.8125</td><td>29 / 16</td></tr>
<tr><td>011110</td><td>1.875</td><td>30 / 16</td></tr>
<tr><td>011111</td><td>1.9375</td><td>31 / 16</td></tr>
</table>
</td></tr>
<tr><td>M0.6</td><td>31:0</td><td>Slot 6 X Pixel Offset</td></tr>
<tr><td>M0.5</td><td>31:0</td><td>Slot 5 X Pixel Offset</td></tr>
<tr><td>M0.4</td><td>31:0</td><td>Slot 4 X Pixel Offset</td></tr>
<tr><td>M0.3</td><td>31:0</td><td>Slot 3 X Pixel Offset</td></tr>
<tr><td>M0.2</td><td>31:0</td><td>Slot 2 X Pixel Offset</td></tr>
<tr><td>M0.1</td><td>31:0</td><td>Slot 1 X Pixel Offset</td></tr>
</table>

DWord	Bit	Description
M0.0	31:0	**Slot 0 X Pixel Offset**
M1.7	31:0	**Slot 15 X Pixel Offset**
M1.6	31:0	**Slot 14 X Pixel Offset**
M1.5	31:0	**Slot 13 X Pixel Offset**
M1.4	31:0	**Slot 12 X Pixel Offset**
M1.3	31:0	**Slot 11 X Pixel Offset**
M1.2	31:0	**Slot 10 X Pixel Offset**
M1.1	31:0	**Slot 9 X Pixel Offset**
M1.0	31:0	**Slot 8 X Pixel Offset**
M2.7	31:0	**Slot 7 Y Pixel Offset** Specifies the Y pixel offset for slot 7. Format = S4 2's complement representing units of 1/16 pixel. The upper 28 bits are ignored. Range = [-8/16, +7/16]
M2.6	31:0	**Slot 6 Y Pixel Offset**
M2.5	31:0	**Slot 5 Y Pixel Offset**
M2.4	31:0	**Slot 4 Y Pixel Offset**
M2.3	31:0	**Slot 3 Y Pixel Offset**
M2.2	31:0	**Slot 2 Y Pixel Offset**
M2.1	31:0	**Slot 1 Y Pixel Offset**
M2.0	31:0	**Slot 0 Y Pixel Offset**
M3.7	31:0	**Slot 15 Y Pixel Offset**
M3.6	31:0	**Slot 14 Y Pixel Offset**

DWord	Bit	Description
M3.5	31:0	**Slot 13 Y Pixel Offset**
M3.4	31:0	**Slot 12 Y Pixel Offset**
M3.3	31:0	**Slot 11 Y Pixel Offset**
M3.2	31:0	**Slot 10 Y Pixel Offset**
M3.1	31:0	**Slot 9 Y Pixel Offset**
M3.0	31:0	**Slot 8 Y Pixel Offset**

Coarse to Pixel Phase Initiating Message (SIMD8 or SIMD16 Message)

The initiating message payload is required, as send message requires at least one GRF register. The payload register may be dummy (to satisfy non-zero payload requirement) or it may include Output Pixel Coverage Mask if coarse pixel shading phase outputs PixelCoverage or discards coarse pixels. The Output Pixel Coverage Mask Valid bit indicates if message payload includes valid data.

Programming Note: This message may be simd8 or simd16 if the thread dispatch rate is simd8. This message MUST be simd16 if the thread dispatch rate is simd16.

M0.7	31:0	Slot 7 Output Pixel Coverage Mask Format = U32
M0.6	31:0	Slot 6 Output Pixel Coverage Mask
M0.5	31:0	Slot 5 Output Pixel Coverage Mask
M0.4	31:0	Slot 4 Output Pixel Coverage Mask
M0.3	31:0	Slot 3 Output Pixel Coverage Mask
M0.2	31:0	Slot 2 Output Pixel Coverage Mask
M0.1	31:0	Slot 1 Output Pixel Coverage Mask
M0.0	31:0	Slot 0 Output Pixel Coverage Mask
M1.x		Slots 15:8 Output Pixel Coverage Mask (delivered only if original dispatch mode was SIMD16)

Writeback Message

SIMD8

The response length for all SIMD8 messages is 2. The data for each slot is written only if its corresponding execution mask bit is set.

DWord	Bit	Description
W0.7	31:0	**Barycentric[1] for Slot 7** Format = IEEE_Float
W0.6	31:0	**Barycentric[1] for Slot 6**
W0.5	31:0	**Barycentric[1] for Slot 5**
W0.4	31:0	**Barycentric[1] for Slot 4**
W0.3	31:0	**Barycentric[1] for Slot 3**
W0.2	31:0	**Barycentric[1] for Slot 2**
W0.1	31:0	**Barycentric[1] for Slot 1**
W0.0	31:0	**Barycentric[1] for Slot 0**
W1.7	31:0	**Barycentric[2] for Slot 7** Format = IEEE_Float
W1.6	31:0	**Barycentric[2] for Slot 6**
W1.5	31:0	**Barycentric[2] for Slot 5**
W1.4	31:0	**Barycentric[2] for Slot 4**
W1.3	31:0	**Barycentric[2] for Slot 3**
W1.2	31:0	**Barycentric[2] for Slot 2**
W1.1	31:0	**Barycentric[2] for Slot 1**
W1.0	31:0	**Barycentric[2] for Slot 0**

SIMD16

The response length for all SIMD16 messages is 4. The data for each slot is written only if its corresponding execution mask bit is set.

DWord	Bit	Description
W0.7	31:0	**Barycentric[1] for Slot 7** Format = IEEE_Float
W0.6	31:0	**Barycentric[1] for Slot 6**
W0.5	31:0	**Barycentric[1] for Slot 5**
W0.4	31:0	**Barycentric[1] for Slot 4**
W0.3	31:0	**Barycentric[1] for Slot 3**
W0.2	31:0	**Barycentric[1] for Slot 2**
W0.1	31:0	**Barycentric[1] for Slot 1**
W0.0	31:0	**Barycentric[1] for Slot 0**
W1.7	31:0	**Barycentric[2] for Slot 7** Format = IEEE_Float
W1.6	31:0	**Barycentric[2] for Slot 6**
W1.5	31:0	**Barycentric[2] for Slot 5**
W1.4	31:0	**Barycentric[2] for Slot 4**
W1.3	31:0	**Barycentric[2] for Slot 3**
W1.2	31:0	**Barycentric[2] for Slot 2**
W1.1	31:0	**Barycentric[2] for Slot 1**
W1.0	31:0	**Barycentric[2] for Slot 0** Format = IEEE_Float
W2.7	31:0	**Barycentric[1] for Slot 15**

DWord	Bit	Description
W2.6	31:0	**Barycentric[1] for Slot 14**
W2.5	31:0	**Barycentric[1] for Slot 13**
W2.4	31:0	**Barycentric[1] for Slot 12**
W2.3	31:0	**Barycentric[1] for Slot 11**
W2.2	31:0	**Barycentric[1] for Slot 10**
W2.1	31:0	**Barycentric[1] for Slot 9**
W2.0	31:0	**Barycentric[1] for Slot 8**
W3.7	31:0	**Barycentric[2] for Slot 15**
W3.6	31:0	**Barycentric[2] for Slot 14**
W3.5	31:0	**Barycentric[2] for Slot 13**
W3.4	31:0	**Barycentric[2] for Slot 12**
W3.3	31:0	**Barycentric[2] for Slot 11**
W3.2	31:0	**Barycentric[2] for Slot 10**
W3.1	31:0	**Barycentric[2] for Slot 9**
W3.0	31:0	**Barycentric[2] for Slot 8**

Coarse to Pixel Mapping Writeback Message (SIMD8 Message)

The writeback message layout is similar to SIMD8 PS thread payload. The first two writeback phases (W0 and W1) are mandatory, and remaining phases are optional as indicated by Coarse to Pixel Mapping Phase Enable bits.

W0.7	31:16	Pixel Mask VMASK (SubSpan[1:0]) : Indicates lit and helper pixels within the two subspans; for each 4-bit group, the OR of the corresponding 4-bit group in the Pixel Mask DMASK. The Pixel Shader kernel should use VMASK as dispatch mask if pixel and/or sample phases compute implicit derivatives. This field must not be modified by the Pixel Shader kernel.
	15:0	Pixel Mask DMASK (SubSpan[1:0]) : Indicates which pixels within the two subspans are lit (rasterized and passed early Z/stencil tests if present). The Pixel Shader kernel should use DMASK as dispatch mask if pixel and/or sample phases do not compute implicit derivatives.
W0.6	31:0	Reserved (MBZ)
W0.5	31:0	Reserved (MBZ)
W0.4	31:0	Reserved (MBZ)
W0.3	31:16	Y1 : Y coordinate (screen space) for upper-left pixel of subspan 1 (slot 4) Format = U16
	15:0	X1 : X coordinate (screen space) for upper-left pixel of subspan 1 (slot 4) Format = U16
W0.2	31:16	Y0 : Y coordinate (screen space) for upper-left pixel of subspan 0 (slot 0) Format = U16
	15:0	X0 : X coordinate (screen space) for upper-left pixel of subspan 0 (slot 0) Format = U16
W0.1	31:0	Reserved (MBZ)
W0.0	31:20	Reserved (MBZ)
	19:16	MSAA rate (multisample count) Format: U4 [1..16] This field specifies MSAA sampling rate (required for PS+S monolithic shader).

	15:4	Reserved (MBZ)
	3:0	Next Pixel Shader Phase Format = U4 [0..15] Pixel Interpolator returns identifier of the next pixel phase to be queried in monolithic CPS+PS(+S) implementation. The next phase may be different than requested phase + 1. The value of 0 indicates there are no more pixels available beyond requested phase. The intended usage scenario is 1) ISA kernel sends Coarse to Pixel Mapping requesting data for phase 0 2) If the returned next pixel phase is zero, ISA kernel executes pixel code for phase 0 (one loop pass) and terminates 3) If the returned next pixel phase is greater than zero, ISA kernel executes pixel code for requested phase and then requests shader input for next phase (until value of zero is returned)
W1.7	31:0	Reserved (MBZ)
W1.6	31:0	Reserved (MBZ)
W1.5	31:0	Reserved (MBZ)
W1.4	31:0	Reserved (MBZ)
W1.3		Parent coarse shader slots for pixel dispatched at slots 7:6
W1.2		Parent coarse shader slots for pixel dispatched at slots 5:4
W1.1		Parent coarse shader slots for pixel dispatched at slots 3:2
W1.0		Parent coarse shader slots for pixel dispatched at slots 1:0
W2.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 7 This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled. Format = IEEE_Float
W2.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 6
W2.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 5
W2.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 4
W2.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 3
W2.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 2
W2.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 1
W2.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 0
W3		Perspective Pixel Location Barycentric[2] for Slots 7:0
W4:5		Perspective Centroid Barycentric, included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.
W6:7		Linear Pixel Location Barycentric, included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.

W8:9		Linear Centroid Barycentric, included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.
W10		Pixel Shader Input Coverage Mask, for slots 7:0 This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled. Format = U32

Coarse to Pixel Mapping Writeback Message (SIMD16 Message)

The writeback message layout is similar to SIMD16 PS thread payload. The first two writeback phases (W0 and W1) are mandatory, and remaining phases are optional as indicated by Coarse to Pixel Mapping Phase Enable bits.

Note: If all optional phases were enabled, the total length of the writeback message would exceed 16 GRFs which is not allowed. In such case, ISA kernel must split request into two PI messages. The following table does not reflect this split, and lists all phases together in the order they are delivered.

W0.7	31:16	Pixel Mask VMASK (SubSpan[1:0]) : Indicates lit and helper pixels within the two subspans; for each 4-bit group, the OR of the corresponding 4-bit group in the Pixel Mask DMASK. The Pixel Shader kernel should use VMASK as dispatch mask if pixel and/or sample phases compute implicit derivatives. This field must not be modified by the Pixel Shader kernel.
	15:0	Pixel Mask DMASK (SubSpan[1:0]) : Indicates which pixels within the two subspans are lit (rasterized and passed early Z/stencil tests if present). The Pixel Shader kernel should use DMASK as dispatch mask if pixel and/or sample phases do not compute implicit derivatives.
W0.6	31:0	Reserved (MBZ)
W0.5	31:16	Y3: Y coordinate (screen space) for upper-left pixel of subspan 3 (slot 12) Format = U16
	15:0	X3: X coordinate (screen space) for upper-left pixel of subspan 3 (slot 12) Format = U16
W0.4	31:16	Y2 : Y coordinate (screen space) for upper-left pixel of subspan 2 (slot 8) Format = U16
	15:0	X2 : X coordinate (screen space) for upper-left pixel of subspan 2 (slot 8) Format = U16

W0.3	31:16	Y1 : Y coordinate (screen space) for upper-left pixel of subspan 1 (slot 4) Format = U16
	15:0	X1 : X coordinate (screen space) for upper-left pixel of subspan 1 (slot 4) Format = U16
W0.2	31:16	Y0 : Y coordinate (screen space) for upper-left pixel of subspan 0 (slot 0) Format = U16
	15:0	X0 : X coordinate (screen space) for upper-left pixel of subspan 0 (slot 0) Format = U16
W0.1	31:0	Reserved (MBZ)
W0.0	31:20	Reserved (MBZ)
	19:16	MSAA rate (multisample count) Format: U4 [1..16] This field specifies MSAA sampling rate (required for PS+S monolithic shader).
	15:4	Reserved (MBZ)
	3:0	Next Pixel Shader Phase Format = U4 [0..15] Pixel Interpolator returns identifier of the next pixel phase to be queried in monolithic CPS+PS(+S) implementation. The next phase may be different than requested phase + 1. The value of 0 indicates there are no more pixels available beyond requested phase. The intended usage scenario is 1) ISA kernel sends Coarse to Pixel Mapping requesting data for phase 0 2) If the returned next pixel phase is zero, ISA kernel executes pixel code for phase 0 (one loop pass) and terminates 3) If the returned next pixel phase is greater than zero, ISA kernel executes pixel code for requested phase and then requests shader input for next phase (until value of zero is returned)
W1.7	32:16	Parent coarse shader slot for pixel dispatched at slot 15 Format: U16 Valid range: 0 - 15

	15:0	Parent coarse shader slot for pixel dispatched at slot 14 Format: U16 Valid range: 0 - 15
W1.6		Parent coarse shader slots for pixel dispatched at slots 13:12
W1.5		Parent coarse shader slots for pixel dispatched at slots 11:10
W1.4		Parent coarse shader slots for pixel dispatched at slots 9:8
W1.3		Parent coarse shader slots for pixel dispatched at slots 7:6
W1.2		Parent coarse shader slots for pixel dispatched at slots 5:4
W1.1		Parent coarse shader slots for pixel dispatched at slots 3:2
W1.0		Parent coarse shader slots for pixel dispatched at slots 1:0
W2.7	31:0	Perspective Pixel Location Barycentric[1] for Slot 7 This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled. Format = IEEE_Float
W2.6	31:0	Perspective Pixel Location Barycentric[1] for Slot 6
W2.5	31:0	Perspective Pixel Location Barycentric[1] for Slot 5
W2.4	31:0	Perspective Pixel Location Barycentric[1] for Slot 4
W2.3	31:0	Perspective Pixel Location Barycentric[1] for Slot 3
W2.2	31:0	Perspective Pixel Location Barycentric[1] for Slot 2
W2.1	31:0	Perspective Pixel Location Barycentric[1] for Slot 1
W2.0	31:0	Perspective Pixel Location Barycentric[1] for Slot 0
W3		Perspective Pixel Location Barycentric[2] for Slots 7:0
W4		Perspective Pixel Location Barycentric[1] for Slots 15:8
W5		Perspective Pixel Location Barycentric[2] for Slots 15:8
W6:9		Perspective Centroid Barycentric, included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.
W10:13		Linear Pixel Location Barycentric, included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.
W14:17		Linear Centroid Barycentric, included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.
W18		Pixel Shader Input Coverage Mask, for slots 7:0 This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled. Format = U32

W19		Pixel Shader Input Coverage Mask, for slots 15:8 Format = U32 This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.

Message Gateway

The Message Gateway has these functions:

- Barrier messages for thread-to-thread synchronization within a thread group.
- Event messages (Monitor, Wait, and Signal) for thread-to-thread signaling across GPU.

End of Thread messages to exit the GPGPU threads.

Messages

Gateway messages are used to signal conditions between threads.

Gateway Message Summary

Message	Description
EOT	Signals the end of this thread. Some 3D threads use a different message for EOT.
Signal_Barrier	Signals that this thread has produced a barrier. When all the threads sharing the barrier have signaled, then all the consumer threads are notified. Consumer threads wait for their notification using the SYNC_BARRIER EU instruction.
Monitor_Event	Specifies this thread is expecting the specified event to be signaled by another thread. A thread can only expect one event. Events cannot be used when a barrier is being used because they use the same notification mechanism.
Monitor_No_Event	Specifies this thread is no longer expecting any events (cancels Monitor_Event). When a thread starts, no events are expected.
Signal_Event	Signals this event to all threads in the GPU. Causes threads waiting on this event to continue executing.
Wait_For_Event	Specifies how long this thread should wait for the next notification of the event specified by Monitor_Event. The thread then waits the notification using SYNC_BARRIER. If the notification does not come within the specified timeout, then a notification is automatically produced.

Barriers are allocated per threadgroup in a subslice. When all the threads in the threadgroup have signaled the barrier, then all the threads receive a notification in their N0.0 register.

Monitors are a general-purpose method for any thread on the GPU to signal to any other cooperative threads on the GPU that an Event has occurred. Because of race conditions, threads use global atomic semaphores for guaranteed synchronization of the events. The signaling of events provides a way to reduce the polling frequency required with atomic memory-based semaphores.

Barrier Messages

Barriers are used to synchronize the execution of two or more concurrently running threads in a threadgroup.

Each thread executes the barrier in two parts:

- send a signal_barrier message to Gateway port
- use sync.bar instruction to wait for a barrier completion notification from Gateway port

When all the threads sharing the same barrier have signaled the barrier, then the Gateway port notifies the threads that the barrier is complete.

The barrier is marked inactive when it is allocated, and whenever a barrier completion notification is sent.

A barrier is allocated when a threadgroup is dispatched and released when all the threads in the threadgroup exit. A thread that signals a barrier must wait for the completion notification before the thread exits.

Producer and Consumer Barrier Messages

Producer_Only and Consumer_Only barrier messages are an extension of the traditional Producer_Consumer barrier operation for a threadgroup. Threads using traditional barriers are both a producer of the barrier, and a consumer of the barrier completion notification.

Barrier Operation	Thread Usage Model
Producer_Only	The producer thread sends signal_barrier message with PRODUCER_ONLY parameter, indicating the thread has produced its data for the consumer threads. The producer thread does not receive the completion notification, and does not use the sync.bar instruction.
Consumer_Only	The consumer thread sends signal_barrier message with CONSUMER_ONLY parameter, indicating the thread wants to be notified when this barrier is completed. The consumer thread then waits for the completion notification to synchronize with the producer threads.
Producer_Consumer	The traditional barrier message sends signal_barrier message with PRODUCER_CONSUMER parameter, indicating the thread has produced its data and that it wants to be notified when the barrier is completed by all threads. The producer_consumer thread then waits for the completion notification to synchronize with all the other threads.

Each barrier must have at least 1 producing thread and 1 consuming thread. This can be formed with any combination of Producer_Consumer, Producer_Only, or Consumer_Only threads. The barrier synchronization is not complete until all the producer and consumer threads for a barrier have sent their signal_barrier message.

Event Messages

Event Monitors are used by threads to be notified of an event signaled by another running threads in the GPU.

When a thread sends a **signal_event** message to Gateway port, the event is broadcast to all running threads in the GPU that have previously requested to be notified of the event.

To receive an event, a thread:

1. Requests to be notified of an Event by sending a monitor_event message to the Gateway port. This registers the thread as expecting the event.
2. Waits until the event is signaled (or a timeout occurs) by sending a wait_event message to the Gateway port.
3. Checks a memory-based semaphore to ensure that the signal was received.

Events are not, by themselves, a thread synchronization mechanism. Typically, independently running threads synchronize using atomic operations on memory-based semaphores. Events are a mechanism to reduce polling operations on memory-based semaphores.

Sample code to send and receive events with a semaphore:

Signaling Thread	Receiving Thread

extern int semaphore = 0;
// do work, produce results ...
atomic_store(&semaphore, 1);
signal_event(&semaphore);
	extern int semaphore;
	monitor_event(&semaphore);
	while ((atomic_cmpxchg(&semaphore, 1, 0) == 0) {
	  wait_event( );
	}
	monitor_no_event( );
	// consume other thread's results ...

When a thread exits, it is automatically unregistered from any events (implicit monitor_no_event).

End of Thread (EOT) Message

A thread exits by executing a send instruction with the EOT bit set. Only a small number of send messages support EOT. The Gateway EOT message is one of those few messages. (Which message to use for a thread depends on which fixed function was used to dispatch the thread.)

The Gateway EOT message removes the barrier and event tracking for the thread and decrements the use count on the thread's barriers and shared local memory. When the use count for a barrier or shared local memory reaches zero, then that resource is available for allocation by another threadgroup.