

Intel® Iris® Xe MAX Graphics Open Source

Programmer's Reference Manual

For the 2020 Discrete GPU formerly named "DG1"

Volume 9: Render Engine

February 2021, 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.

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.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Table of Contents

Render Engine	1
Workload Submission	1
Context Submission Overview.....	1
Render-3D-GPGPU Command Streamer	2
MI Commands Supported by POCS.....	20
Compute Command Streamer	38
Engine State.....	48
Software Interface	65
3D Pipeline Stages.....	72
3D Pipeline-Level State	73
3D Pipeline Geometry.....	76
3D Pipeline Rasterization.....	184
Pixel.....	254
COLOR_CALC_STATE.....	315
3DSTATE_BLEND_STATE_POINTERS.....	315
3DSTATE_DEPTH_STENCIL_STATE_POINTERS.....	315
DEPTH_STENCIL_STATE.....	315
BLEND_STATE.....	315
CC_VIEWPORT.....	316
Statistics Gathering.....	316
GPGPU Compute Pipeline.....	316
General Purpose Compute Model.....	316
GPGPU Context in GPU Hardware	319
Programming the GPGPU Pipeline	320
Commands for GPGPU Pipe.....	327
MEDIA_VFE_STATE	328
MEDIA_STATE_FLUSH	329
Thread Spawner (TS)	338
Thread Dispatch	349
Thread Tracking and Synchronization.....	355
Context Switch for GPGPU and Media.....	358
3D and GPGPU Programs	360

EU Overview.....	360
Integer Numeric Data Types	441
Floating-Point Numeric Data Types.....	443
Packed Signed Half-Byte Integer Vector.....	445
Packed UnSigned Half-Byte Integer Vector.....	446
Packed Restricted Float Vector	447
IEEE Floating Point Mode.....	450
Alternative Floating Point Mode	453
IEEE Floating-Point Exceptions.....	460
Floating-Point Compare Operations.....	465
Float to Integer.....	469
Integer to Integer with Same or Higher Precision.....	469
Integer to Integer with Lower Precision	470
Integer to Float.....	470
Double Precision Float to Single Precision Float	470
Single Precision Float to Double Precision Float	471
Invoking the System Routine.....	474
Returning to the Application Thread.....	475
System IP (SIP)	475
System Routine Register Space	475
System Scratch Memory Space.....	476
Conditional Instructions Within the System Routine	476
Illegal Opcode.....	477
Undefined Opcodes.....	477
Software Exception	477
Context Save and Restore	478
Illegal Instruction Format	478
Malformed Message	478
GRF Register Out of Bounds	479
Hung Thread.....	479
Instruction Fetch Out of Bounds	479
FPU Math Errors	479
Computational Overflow	479
SIMD Instructions and SIMD Width	482

Instruction Operands and Register Regions.....	483
Instruction Execution.....	483
EU Compact Instruction Format.....	484
Compact Instructions Format.....	484
EU Instruction Compaction Tables.....	487
Move and Logic Instructions.....	499
Flow Control Instructions.....	500
Miscellaneous Instructions.....	501
Parallel Arithmetic Instructions.....	501
Vector Arithmetic Instructions.....	502
Special Instructions.....	502
Instruction Groups.....	503
Destination Register.....	505
Source Register.....	506
Address Registers.....	507
Register Files and Register Numbers.....	507
Relative Location and Stack Control.....	508
Regions.....	508
Types.....	509
Write Mask.....	509
Swizzle Control.....	509
Immediate Values.....	509
Predication and Modifiers.....	510
Instruction Options.....	511
Grammar.....	518
IGA Grammar.....	518
Grammar.....	527
IGA Grammar.....	527
Load-Store Pseudo Instructions.....	538
Syntax.....	538
Loads.....	538
Stores.....	538
Operand Syntax.....	539
Examples.....	540



Supported Messages	540
Split-Sends and Conditional Sends.....	551
Operand Mapping.....	551
Load Pseudo-Instructions.....	551
Block Messages.....	551
Vector Messages with a Header.....	552
Headerless Vector Messages.....	552
Store Pseudo-Instructions.....	552
Block Messages.....	553
Vector Messages with a Header.....	554
Headerless Vector Messages.....	554
Round Instructions	558
rddd - Round Down.....	559
rnde - Round to Nearest or Even.....	560
rndu - Round Up.....	561
rndz - Round to Zero.....	562
math - Extended Math Function.....	563
INV - Inverse.....	563
LOG - Logarithm.....	563
EXP - Exponent.....	563
SQRT - Square Root.....	563
RSQ - Reciprocal Square Root.....	564
SIN - SINE.....	564
COS - COSINE.....	564
INT DIV - Integer Divide.....	564
INVM/RSQRTM.....	565
SEND Instruction.....	569
EU Instructions.....	571
SEND Instructions.....	573
Control Flow Instructions.....	574
Shared Functions.....	575
Sampler SW Performance Recommendations.....	582
TEXCOORDMODE_MIRROR_101.....	601
Procedural Textures and Texel Shading.....	621

3D Sampler Message Types	626
gather4 Message Types	637
Definitions.....	637
Supported Variants:.....	637
Restrictions and Programming Notes for gather4:.....	638
Restrictions and Programming Notes for gather4_c:	638
Restrictions and Programming Notes for gather4_po:	639
Restrictions and Programming Notes for gather4_po_c:	639
sampleinfo Message Type	639
Supported Variants:.....	640
Restrictions and Programming Notes for sampleinfo:.....	640
LOD Message Type.....	640
LOD Message Definition	640
Supported Variants:.....	640
Restrictions and Programming Notes for LOD:	640
resinfo Message Type	641
Supported Variants:.....	642
Restrictions and Programming Notes for resinfo:	642
cache_flush Message Type.....	642
cache_flush Message Definition.....	642
Supported Variants:.....	642
Media Message Types	643
sample_unorm Message Types.....	643
Supported Variants:.....	643
Restrictions and Programming Notes for sample_unorm, sample_unorm_RG, sample_unorm_killpix, sample_unorm_RG_killpix:	644
sample_8x8 Message Type	644
Supported Variants:.....	645
Restrictions and Programming Notes for sample_8x8:.....	645
Message Format	645
HWord Aligned Block Read/Write Messages.....	702
Word Untyped Atomic Integer Messages	707
Word Untyped Atomic Float Messages.....	713
Word Atomic Counter Messages	718



DWord URB Read/Write Messages	726
Message-Specific Descriptors	749
Render Target Write Message.....	749
Replicate Data.....	756
Single Source.....	756
Dual Source.....	756
Message Data Payloads.....	757
Render Target Data Payloads	758
Message Payload.....	776
Writeback Message to Requester Thread.....	777
Broadcast Writeback Message	777

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 work submission to the Rendering engine which can run 3D, Compute and Programmable Media workloads

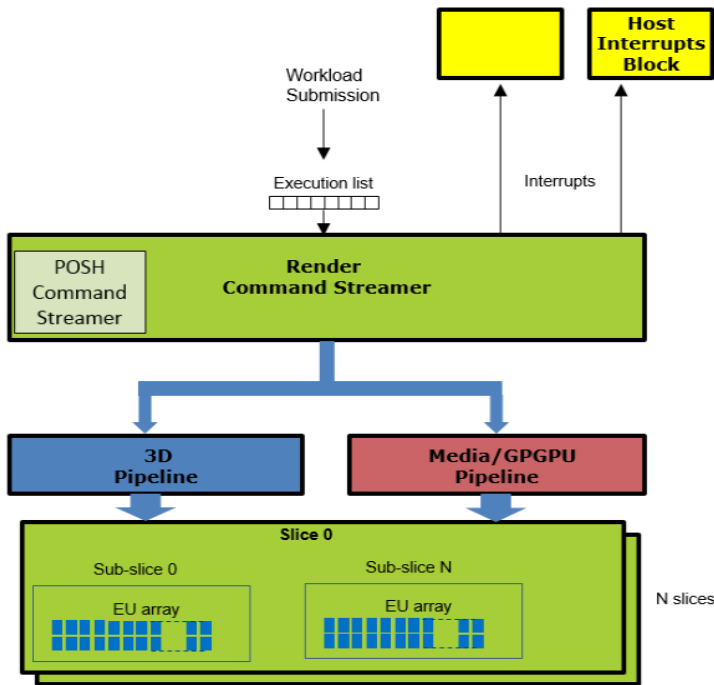
Context Submission Overview

Work into the Render/GPGPU engine is fed using the Render Command Streamer.

The Render engine runs in one of the following modes (that is specified using the PIPE_SELECT command):

- 3D
- Media/GPGPU

When Software submits multiple elements(contexts) into the execution list, the hardware executes the elements serially.



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.

POCS and RCS Synchronization

Once POCS is triggered, it executes parallel to RCS, it only stops (doesn't switch out) when it runs out of command (head equals to tail) or on encountering unsuccessful semaphore wait. Command sequence execution of POCS is completely asynchronous to RCS command sequence execution. SW is responsible to explicitly synchronize POCS and RCS command sequence execution whenever required based on the various produce consume model using MI_SEMAPHORE_WAIT command.

GPGPU/Media

POSH pipe supports parallel execution during GPGPU and Media workloads under the condition that the L3 is not reconfigured and the GPGPU/Media workload will not use more than 32KB of the URB. Currently the POSH and GPGPU on the RCS share the same URB space. However, the first 32KB are not used when POSH is only running.

Reconfiguring the URB cannot occur when switching between 3D and Compute/Media workloads if it is possible for POSH pipeline to be active. SW can ensure POSH is inactive using synchronization commands in the case POSH was enabled earlier in the context.

Mid-thread preemption must be disabled if it is possible for POSH to be running in parallel with compute workloads.

HW Binding Table with RS Disabled

RCS sets up the HW Binding Table functionality when 3DSTATE_BTP_POOL_ALLOC is programmed with RS disabled. POSH pipe uses the mode set by RCS. SW will explicitly synchronize POCS and RCS to ensure they always work in the same mode of operation wither HW BTP Enabled or Disabled. Note that both POCS and RCS will maintain their own copies of 3DSTATE_BTP_POOL_ALLOC.

Protection-On/Off Mode

RenderCS controls the Protection-On/Off mode at all times for both POSH and Render pipes. Protection-on/off mode set by RenderCS applies to memory clients form both render pipe and POSH pipe based on the protection on signal from RCS. SW must explicitly ensure both POSH and Render pipes are synchronized around Protection and ProtectionOff zones during the command sequencing.

Render Engine Command Streamer (RCS)

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
<ul style="list-style-type: none"> • 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.
<ul style="list-style-type: none"> • 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

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
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
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

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
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
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	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

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
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	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	57h-59h	Reserved	
0h	60h-68h	Reserved	
0h	69h	Reserved	
0h	6Eh	Reserved	
0h	74h	Reserved	
0h	75h	Reserved	
0h	76h	Reserved	
0h	77h-82h	Reserved	
0h	83h	Reserved	
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

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
1h	11h	3DSTATE_MONOFILTER_SIZE	Sampling Engine
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	3DSTATE_PUSH_CONSTANT_ALLOC_PS	Pixel Shader
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	Reserved	
3h	02h	Reserved	
3h	03h-FFh	Reserved	
4h-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	Reserved	
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:

¹The 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
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**.

Batch Buffer Privilege Register

FORCE_TO_NONPRIV - FORCE_TO_NONPRIV



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
Hardware Status Page Layout

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
DISPLAY_MESSAGE_FORWARD_STATUS - Display Message Forward Status Register
R_PWR_CLK_STATE - Render Power Clock State Register

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

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_MATH
MI_REPORT_HEAD
MI_STORE_DATA_IMM
MI_ATOMIC
MI_COPY_MEM_MEM

Commands
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

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.

Position Only Shader Command Streamer (POCS)

Position only shader (POSH) is a secondary geometry pipeline that has the optional ability to execute the position only vertex shaders and perform the visibility test on these vertices before the actual vertex shader is executed. POSH pipe can run ahead of the original geometry pipe by executing position only vertex shaders and doing visibility test on these vertices and recording this information. Geometry pipe when processing the vertices will use this visibility information outputted by POSH pipe to skip the vertex fetch and shading for vertices that are already marked as culled.

POSH pipe has its own command streamer called Position only command streamer (POCS). A context running on render pipe can exercise POSH capabilities through Render Command Streamer (RCS). RCS manages the POSH pipe through POCS for POSH enabled contexts. Render command streamer loads the context to execute on POCS when a POSH enable context execution begins in render pipe, similarly, preempts context executing in POCS when the POSH enabled context switches out of render pipe. Once POCS is loaded with context it starts executing the ring buffer similar to RCS, refer Programming Model section for more details.

Position Only Command Streamer (POCS)

The POCS (Position Only Shader Command Streamer) unit primarily serves as the programming interface between the render command streamer and the POSH pipe. It is responsible for fetching, decoding, and dispatching of data packets (3D Commands with the header DWord removed) for the POSH pipe.

Logic Functions Included

- MMIO register programming interface.
- DMA action for fetching of ring buffer and batch buffer data from memory.
- Management of the Head pointer for the Ring Buffer.
- Decode and execution of command programmed in ring buffer and batch buffers.
- Flushing the POSH pipe.

Handle NOOP.

The POCS unit only claims memory mapped I/O cycles that are targeted to its range of 0x1_8000 to 0x1_9FFF. The POCS and RCS use semaphore to synchronize their operations.

POCS ExecList Scheduling

Position Only 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)
3DSTATE_PTBR_TILE_PASS_INFO	AP

Table Notes:

AP - Allow Preemption if arbitration is enabled.

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

POSH Programming Model

The POSH + Render pipeline will appear as a monolithic engine from SW perspective. Render Command Streamer (RCS) is hardware front end interface to the SW for the modified Render + POSH pipeline. SW will use a single context (and associated LRCA) to submit work to the modified Render + POSH pipeline through its associated ring buffer.

Context submission model should be visualized as context submitted to RCS. RCS will set up the context definition in HW and triggers POSH pipe to execute the same context, resulting in execution of the same ring buffer by render pipe and POSH pipe in parallel. POSH pipe has its own command streamer called POCS (POSH Command Streamer). Similarly, when the context is switched out on the render pipe due to whatever reasons (Wait For Event, Semaphore Wait or Preemption due to pending execlist), RCS will ensure POSH pipe is preempted and its corresponding logic state is saved through POCS.

POCS and RCS get to see the same ring buffer, however the execution of the same ring buffer by POCS and RCS are asynchronous to each other and its SW responsibility to ensure POCS and RCS are

synchronized through semaphores as and when required. SW will provide independent command buffers (batch buffers) to be executed by RCS and POCS. Marking of batch buffers for POCS and RCS and execution of ring buffer are detailed in the latter subsections.

This model of execution has the following implications:

- POCS and RCS have to run on the same context definition. RCS sets up context with GAM and POCS runs within this address space.
- Even though the currently running context may not be utilizing the POSH pipe, a waiting context with POSH enabled has to wait for the current context to be evicted. (waiting context cannot take advantage of the idle POSH pipe ahead of getting scheduled on the render engine)

POSH Enabled Context

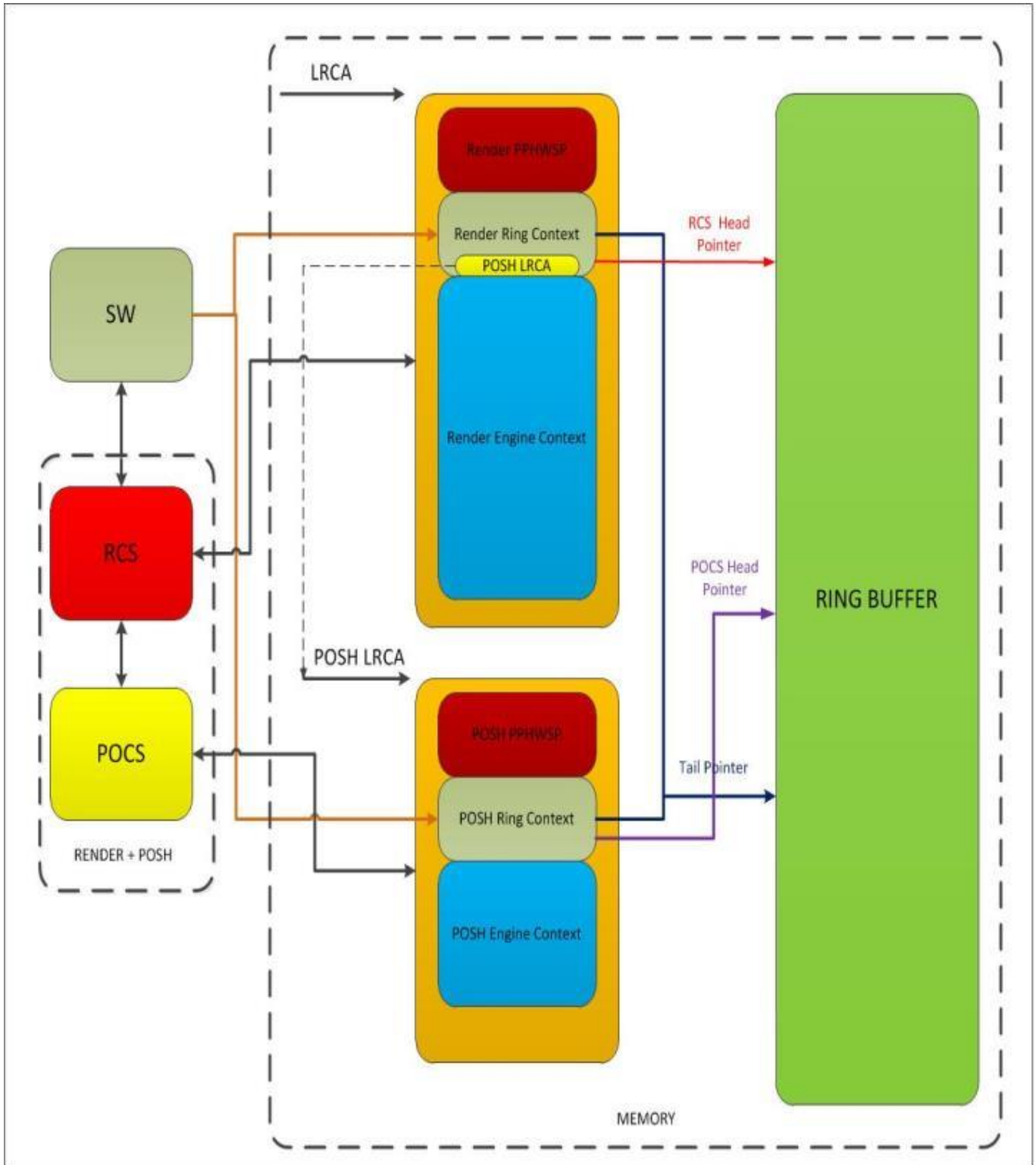
A context submitted to render engine exercising POSH functionality is called "POSH Enabled" context. Application (UMD) decides if a context is POSH Enabled at the time of context creation. A context is indicated as POSH enabled to HW by setting "POSH Enable" bit in CTX_SR_CTL register of RCS. SW allocates additional separate memory space (POSH LRCA) for the POSH Enabled contexts. POSH pipe uses the POSH LRCA for its context state management.

Context Submission and LRCA for POSH

SW will continue to submit POSH enabled contexts to ELSP in RCS. There is no change in the pending execlist submission or context switch status report mechanism to/from RCS.

Listed below are the SW changes required for submission of the POSH enabled context:

- "POSH Enable" bit in CTX_SR_CTL of RCS must be set to indicate POSH enabled context to HW. Refer POSH functionality control section for the bit definition and programming.
- POSH LRCA is provided to RCS through register programming in the ring context of RCS. Refer RCS ring context details below.
- POSH LRCA format is similar to that of RCS, i.e PPHWSP followed by ring context followed by the engine context. However POSH ring context will only have the ring buffer and batch buffer details. POSH ring context will not have the page directory pointers details as the PPGTT is setup by RCS.
- SW does not control POCS context ID independently. The context ID for POCS will be supplied from RCS, and thus will be the same.
- SW must update the ring context of POSH with ring buffer details on the very first submission and whenever the ring buffer start address, control and head pointer details are updated. POSH pipe (POCS) will sample the tail pointer from RCS. Note that the POCS and RCS share the same ring buffer.





POSH LRCA in RCS Ring Context

The table below highlights the POSH LRCA details in RCS ring context. Ring context listed below is for illustration of the change, "Register State Context" in "Render Logical Context Data" should be referred as the final format for implementation.

Description	Unit	# of DW
NOOP	CSEL	1
MI_LOAD_REGISTER_IMM	CSEL	1
Ring Buffer Head	CSEL	2
Ring Tail Pointer Register	CSEL	2
RING_BUFFER_START	CSEL	2
RING_BUFFER_CONTROL	CSEL	2
Batch Buffer Current Head Register (UDW)	CSEL	2
Batch Buffer Current Head Register	CSEL	2
Batch Buffer State Register	CSEL	2
SECOND_BB_ADDR_UDW	CSEL	2
SECOND_BB_ADDR	CSEL	2
SECOND_BB_STATE	CSEL	2
BB_PER_CTX_PTR	CSEL	2
RCS_INDIRECT_CTX	CSEL	2
RCS_INDIRECT_CTX_OFFSET	CSEL	2
NOOP	CSEL	2
NOOP	CSEL	1
MI_LOAD_REGISTER_IMM	CSEL	1
CTX_TIMESTAMP	CSEL	2
PDP3_UDW	CSEL	2
PDP3_LDW	CSEL	2
PDP2_UDW	CSEL	2
PDP2_LDW	CSEL	2
PDP1_UDW	CSEL	2
PDP1_LDW	CSEL	2
PDP0_UDW	CSEL	2
PDP0_LDW	CSEL	2
MI_LOAD_REGISTER_IMM	CSEL	1
POSH_LRCA	CSEL	2
NOOP	CSEL	9
NOOP	CSEL	1
MI_LOAD_REGISTER_IMM	CSEL	1

Description	Unit	# of DW
R_PWR_CLK_STATE	CSEL	2
GPGPU_CSR_BASE_ADDRESS	CSEL	3
NOOP	CSEL	9

POCS Ring Context

Table below details the POSH ring context. Ring context listed below is for illustration of the change, "Register State Context" in "Render Logical Context Data" should be referred as the final format for implementation.

Description	Unit	# of DW
NOOP	CSEL	1
MI_LOAD_REGISTER_IMM	CSEL	1
Ring Buffer Head	CSEL	2
Ring Tail Pointer Register	CSEL	2
RING_BUFFER_START	CSEL	2
RING_BUFFER_CONTROL	CSEL	2
Batch Buffer Current Head Register (UDW)	CSEL	2
Batch Buffer Current Head Register	CSEL	2
Batch Buffer State Register	CSEL	2
SECOND_BB_ADDR_UDW	CSEL	2
SECOND_BB_ADDR	CSEL	2
SECOND_BB_STATE	CSEL	2
BB_PER_CTX_PTR	CSEL	2
RCS_INDIRECT_CTX(Always Invalid)	CSEL	2
RCS_INDIRECT_CTX_OFFSET	CSEL	2
NOOP	CSEL	2



Description	Unit	# of DW
NOOP	CSEL	48

POSH Command Transport

The following subtopics describe the command transport mechanism from SW to POCS.

"POSH Start" Batch Buffers

Batch buffers dedicated to be executed by POSH pipe are indicated by setting the field "POSH Start" in the MI_BATCH_BUFFER_START command header. Once "POSH Start" is set in a batch buffer all the following chained batch buffers and next level batch buffers will implicitly inherit the "POSH Start" field value. Once "POSH Start" is set in a batch buffer all the following command sequences are to be **executed** by POCS until the corresponding batch buffer sequencing is terminated through MI_BATCH_BUFFER_END/MI_CONDITIONAL_BATCH_BUFFER_END command.

Example:

- Once "POSH Start" is encountered in a first level batch buffer by HW, it will get reset only when the first level batch buffer execution is terminated through batch buffer end and the command execution sequence goes back to the ring buffer,
- Similarly, once "POSH Start" is encountered in a second level batch buffer by HW, it will get reset only when the second level batch buffer execution is terminated through batch buffer end and the command execution sequence goes back to the first level buffer,
- Similarly, once when "POSH Start" is encountered in a third level batch buffer by HW, it will get reset only when the third level batch buffer execution is terminated through batch buffer end and the command execution sequence goes back to the second level batch buffer.

Command sequences executed from the "POSH Start" batch buffer may lead to chained batch buffers or next level batch buffers. Batch buffers executed by POCS may have MI Commands, 3DSATE commands and 3DPRIMITIVE commands for POSH pipe, however these will be a subset of the commands that are supported by render pipe. RCS on parsing MI_BATCH_BUFFER_START command with "POSH Start" enabled NOOPS the command and moves on the following command.

MI Commands Supported by POCS

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

POCS doesn't support below commands and SW must not program them as part of the POSH command sequence.

Commands not supported in POSH executed command buffers:

Column Title1
MI_DISPLAY_FLIP
MI_LOAD_SCANLINES_INCL/EXCL
MI_WAIT_FOR_EVENT
MI_USER_INTERRUPT
MI_REPORT_PERF_COUNT
MI_SET_CONTEXT
MI_ARB_ON_OFF

POCS can semaphore signal RCS and vice-versa, however RCS will not forward semaphore signals received from POCS.

3D State Commands Supported by POCS

The table below lists the 3DSTATE Commands Supported by POSH Pipe. State commands programmed for POSH which are not listed in the table below will be gracefully discarded (NOOP'd) by POCS.

3D State Commands
<ul style="list-style-type: none"> • 3DSTATE_VF • 3DSTATE_INDEX_BUFFER • 3DSTATE_VERTEX_BUFFER • 3DSTATE_VERTEX_ELEMENTS • 3DSTATE_VF_COMPONENT_PACKING • 3DSTATE_VF_INSTANCING • 3DSTATE_VF_SGVS • 3DSTATE_VF_TOPOLOGY • 3DSTATE_VF_STATISTICS • 3DPRIMITIVE • 3DSTATE_VS • 3DSTATE_PUSH_CONSTANT_ALLOC_VS • 3DSTATE_CONSTANT_VS • 3DSTATE_BINDING_TABLE_POOL_ALLOC • 3DSTATE_BINDING_TABLE_POINTERS_VS • 3DSTATE_SAMPLER_STATE_POINTERS_VS • 3DSTATE_URB_VS • 3DSTATE_CLIP • 3DSTATE_SFFE • 3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP • 3DSTATE_SCISSOR_STATE_POINTERS

3D State Commands
<ul style="list-style-type: none"> • 3DSTATE_MULTISAMPLE • 3DSTATE_RASTER • 3DSTATE_DRAWING_RECTANGLE • 3DSTATE_INT • PIPECONTROL Command • 3DSTATE_SBE (for PID computation)
<ul style="list-style-type: none"> • 3DSTATE_PRIMITIVE_REPLICATION
<ul style="list-style-type: none"> • 3DSTATE_PTBR_PAGE_POOL_BASE_ADDRESS • 3DSTATE_PTBR_FREE_LIST_BASE_ADDRESS • 3DSTATE_PTBR_RENDER_LIST_BASE_ADDRESS • 3DSTATE_PTBR_TILE_PASS_INFO
Common Non-Pipeline State Commands
<ul style="list-style-type: none"> • STATE_BASE_ADDRESS

"POSH Enable" Batch Buffers

POCS parses/traverses (doesn't execute) the ring buffer to look for batch buffers programmed with "POSH Start" field set. "POSH Enable" field in the MI_BATCH_BUFFER_START command is a hint to POCS to traverse (parse, don't execute) the batch buffer to look for "POSH Start" batch buffers. "POSH Enable" field is only inherited to the chained batch buffer and doesn't get inherit to the next level batch buffers unlike "POSH Start" field. "POSH Enable" field must be explicitly set in the MI_BATCH_BUFFER_START command which calls the next level batch buffers in order for the POCS to parse them to look for "POSH Start" batch buffers. POCS ends the "POSH Enable" batch buffer on executing MI_BATCH_BUFFER_END or on MI_CONDITIONAL_BATCH_BUFFER_END meeting the required condition. "POSH Start" field takes precedence over the "POSH Enable" field in POCS.

Example:

- Once "POSH Enable" is encountered in a first level batch buffer, POCS will traverse the whole of the first level batch buffers (including chained first level) to check for "POSH Start" field in MI_BATCH_BUFFER_START command. POCS by default will not traverse the second level batch buffers. SW must explicitly set the "POSH Enable" field for the second level batch buffer called from first level batch buffer if the second level batch buffer have to be traversed by POCS.
- Similarly, Once "POSH Enable" is encountered in a second level batch buffer, POCS will traverse the whole of the second level batch buffers (including chained second level) to check for "POSH Start" field in MI_BATCH_BUFFER_START command. POCS by default will not traverse the third level batch buffers. SW must explicitly set the "POSH Enable" field for the third level batch buffer called from second level batch buffer if the third level batch buffer have to be traversed by POCS.

- Similarly, Once "POSH Enable" is encountered in a third level batch buffer, POCS will traverse the whole of the third level batch buffers (including chained second level) to check for "POSH Start" field in MI_BATCH_BUFFER_START command.

RCS ignores "POSH Enable" field and has no implications due to the "POSH Enable" field set in the MI_BATCH_BUFFER_START command.

POSH Ring Buffer

POCS and RCS share the same ring buffer. POCS parses the ring buffer to look for batch buffers start commands with "POSH Enable" or "POSH Start" fields set, it doesn't execute any commands programmed in the ring buffer. POCS and RCS executing the same ring buffer results in two different command sequences based on the "POSH Start" and "POSH Enable" fields programmed in various batch buffers.

Examples for Command Buffer Execution by POCS and RCS

There are various possible combinations of command sequences programmable based on the below fields. Example below states few scenarios.

- POSH Start
- POSH Enable
- First Level Batch Buffers
- Second Level Batch Buffers
- Third Level Batch Buffers
- Chaining of First Level Batch Buffers
- Chaining of Second Level Batch Buffers
- Chaining of Third Level Batch Buffers

Acronyms used in the illustrative example below:

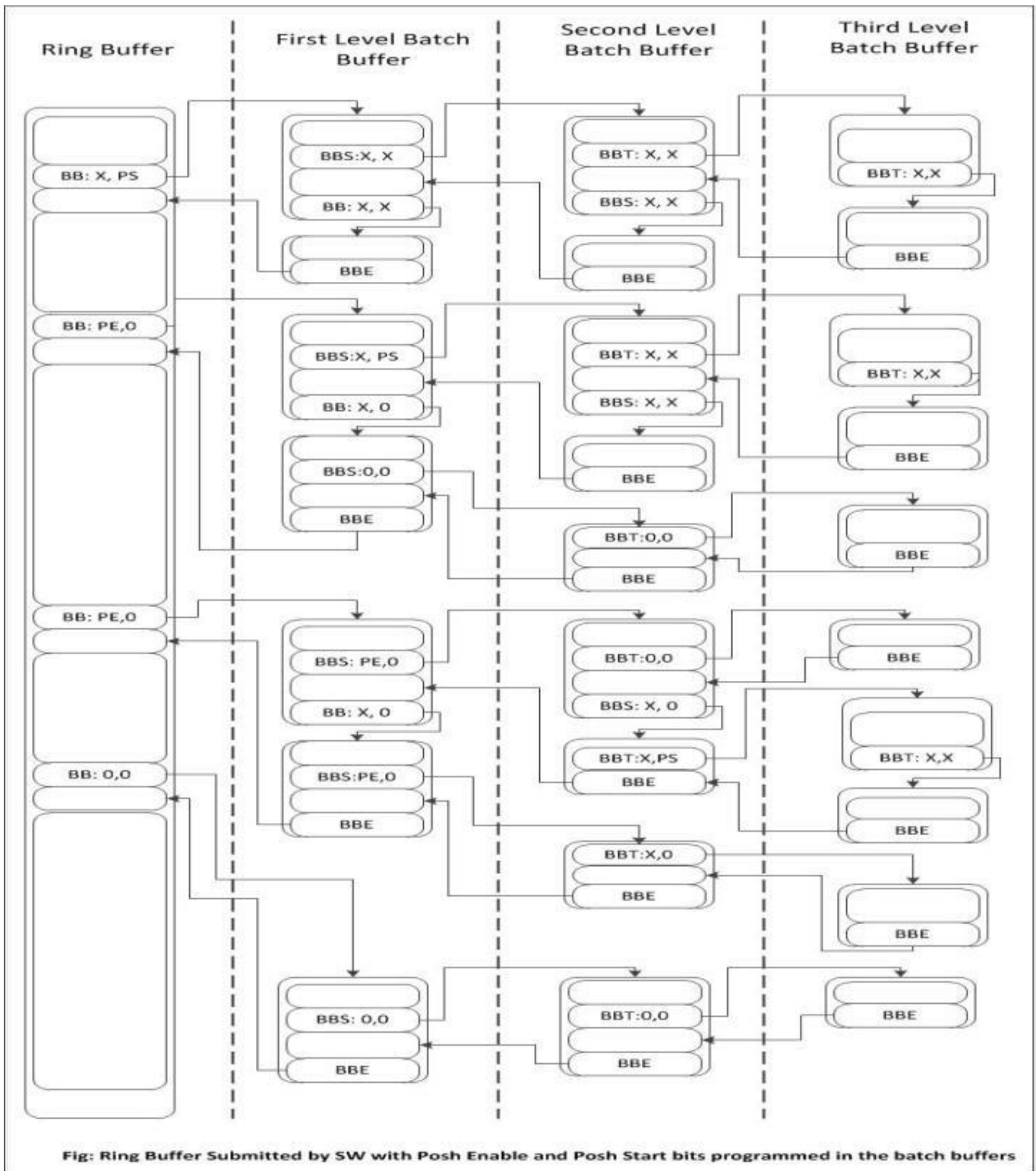
BB: Batch Buffer; **BBS**: Second Level batch buffer; **BBT**: Third Level batch buffer; **BBE**: Batch Buffer End; **PS**: POSH Start; **PE**: POSH Enable; **X**: Field programmed is don't care by HW

In the figures below:

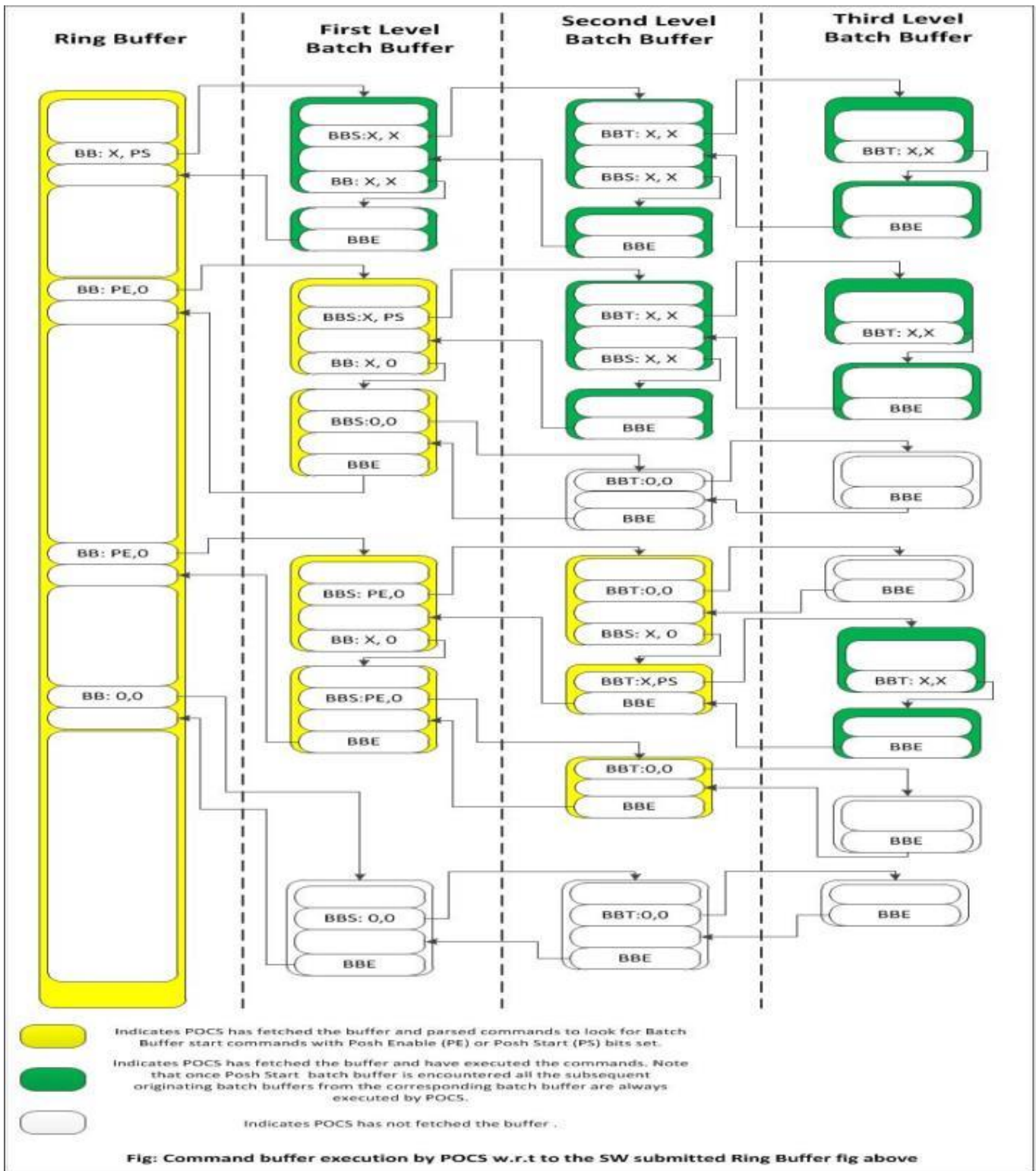
- First figure depicts the command sequence programmed in the ring buffer that is submitted to RCS & POCS.
- Second figure depicts the command sequence outputted by POCS on executing the ring buffer submitted by SW.
- Third figure depicts the command sequence outputted by RCS on executing the ring buffer submitted by SW.

Note that the same ring buffer results in different command sequence execution from POCS and RCS.

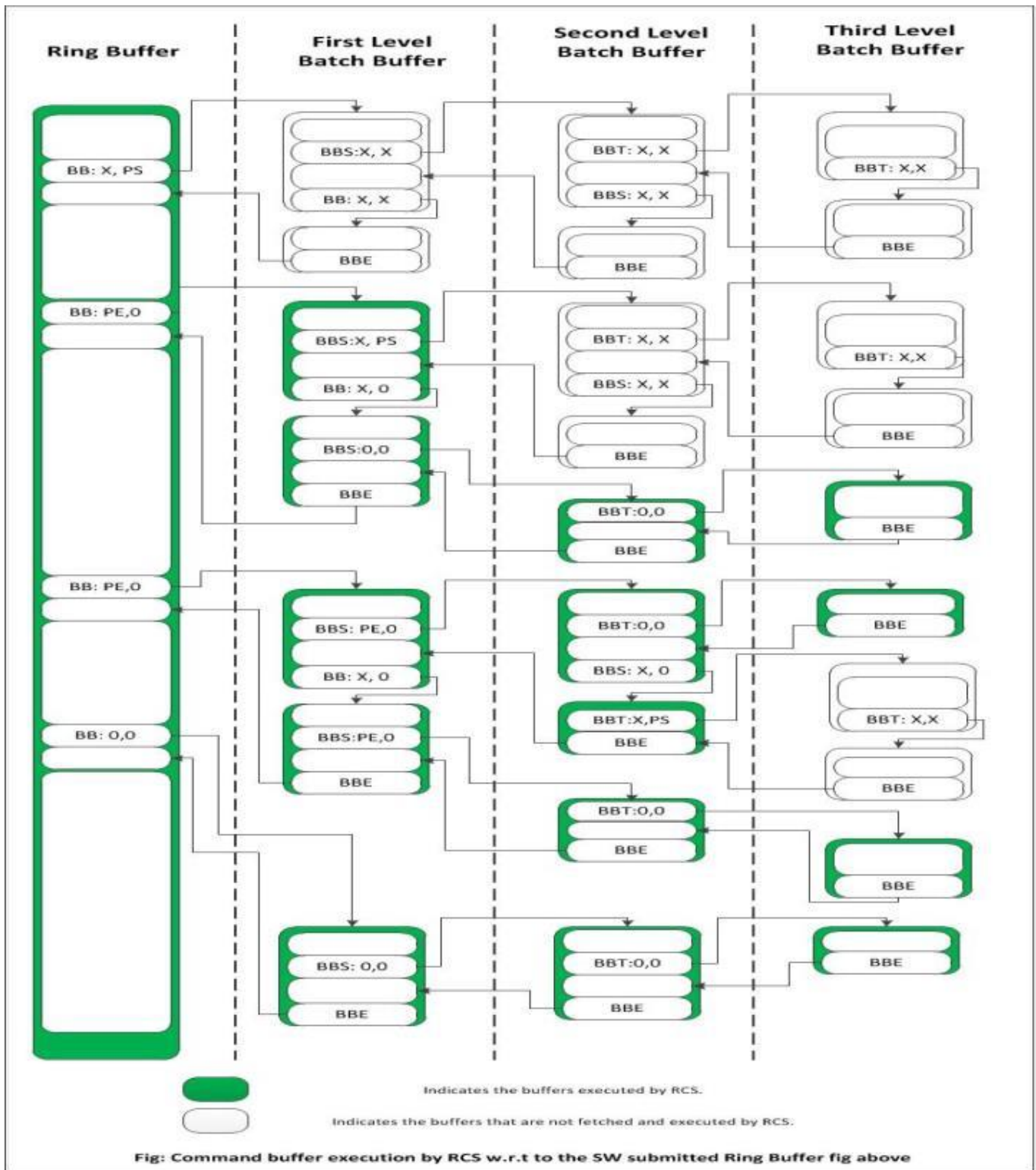
The figure below depicts the command sequence programmed in the ring buffer that is submitted to RCS & POCS.



The figure below depicts the command sequence outputted by POCS on executing the ring buffer submitted by SW in the above diagram.



The figure below depicts the command sequence outputted by RCS on executing the ring buffer submitted by SW.



POSH Preemption

Once the context is loaded to POCS, only way it can be switched out is through explicit preemption from RCS, POCS doesn't switch out an context on encountering un-successful Wait for Events or Semaphore Wait or running out of commands on head equal to tail pointer. RCS on switching out the context either due to synchronous context switch or preemption, it also preempts POCS if the context is POSH enabled. POCS receives preemption from RCS and triggers the preemption flow for POSH pipe. POSH pipe supports 3D object level preemption. Preemption from RCS can happen when POCS is in one of the below states:

- POCSFE has executed the context and have Head Equals Tail.
- POCSFE is busy executing commands.

POSH MMIO

POSH pipe implements its own set of MMIO registers similar to render pipe, however POSH pipe implements the registers relevant to the functionality supported in POSH pipeline. Listed below are the only registers that are accessible in POSH pipeline.

Table 1: Registers in POCSFE

MMIO SYMBOL	Suffix
DMA_FADD	POCS
ACTHD	POCS
ACTHD_UDW	POCS
CS_ALU_ACCU	POCS
CS_ALU_CF	POCS
CS_ALU_SRCA	POCS
CS_ALU_SRCB	POCS
CS_ALU_ZF	POCS
BB_ADDR	POCS
BB_ADDR_DIFF	POCS
BB_ADDR_UDW	POCS
BB_OFFSET	POCS
BB_PER_CTX_PTR	POCS
BB_PREEMPT_ADDR	POCS
BB_PREEMPT_ADDR_UDW	POCS
BB_START_ADDR	POCS
BB_START_ADDR_UDW	POCS
BB_STATE	POCS
CCID	POCS
CTXT_PREMP_DBG	POCS



MMIO SYMBOL	Suffix
CTXT_SR_CTL	POCS
CXT_EL_OFFSET	POCS
CMD_CCTL_0	POCS
RCS_CTXID_PREEMPTION_HINT	POCS
CTX_TIMESTAMP	POCS
CTX_WA_BB_ADDR	POCS
EXCC	POCS
FORCE_TO_NONPRIV_0	POCS
FORCE_TO_NONPRIV_1	POCS
FORCE_TO_NONPRIV_2	POCS
FORCE_TO_NONPRIV_3	POCS
FORCE_TO_NONPRIV_4	POCS
FORCE_TO_NONPRIV_5	POCS
FORCE_TO_NONPRIV_6	POCS
FORCE_TO_NONPRIV_7	POCS
FORCE_TO_NONPRIV_8	POCS
FORCE_TO_NONPRIV_9	POCS
FORCE_TO_NONPRIV_10	POCS
FORCE_TO_NONPRIV_11	POCS
CS_GPR_R_0	POCS
CS_GPR_R_1	POCS
CS_GPR_R_2	POCS
CS_GPR_R_3	POCS
CS_GPR_R_4	POCS
CS_GPR_R_5	POCS
CS_GPR_R_6	POCS
CS_GPR_R_7	POCS
CS_GPR_R_8	POCS
CS_GPR_R_9	POCS
CS_GPR_R_10	POCS
CS_GPR_R_11	POCS
CS_GPR_R_12	POCS
CS_GPR_R_13	POCS
CS_GPR_R_14	POCS
CS_GPR_R_15	POCS
GFX_MODE	POCS
HWS_PGA	POCS

MMIO SYMBOL	Suffix
PWRCTX_MAXCNT	POCS
IPEHR	POCS
IDLEDLY	POCS
CSCMDOP	POCS
CSCMDVLD	POCS
INSTPM	POCS
INSTPS	POCS
MI_PREDICATE_RESULT_1	POCS
MI_PREDICATE_RESULT_2	POCS
MI_MODE	POCS
NOPID	POCS
PDP0	POCS
PDP1	POCS
PDP2	POCS
PDP3	POCS
PR_CTR_THRSH	POCS
PREEMPTDLY	POCS
PREEMPTION_HINT	POCS
PREEMPTION_HINT_UDW	POCS
DMA_FADD_P_UDW	POCS
RING_BUFFER_CTL	POCS
RING_BUFFER_HEAD	POCS
RING_BUFFER_HEAD_PREEMPT_REG	POCS
RING_BUFFER_START	POCS
RING_BUFFER_TAIL	POCS
TIMESTAMP	POCS
RESET_CTRL	POCS
SBB_ADDR	POCS
SBB_ADDR_UDW	POCS
SBB_PREEMPT_ADDR	POCS
SBB_PREEMPT_ADDR_UDW	POCS
SBB_STATE	POCS
SEMA_WAIT_POLL	POCS
CURRENT_LRCA	POCS

Table 2: Registers in POCSBE

MMIO SYMBOL	Suffix
-------------	--------



MMIO SYMBOL	Suffix
3DPRIM_BASE_VERTEX	POCS
3DPRIM_END_OFFSET	POCS
3DPIM_INSTANCE_COUNT	POCS
3DPRIM_START_INSTANCE	POCS
3DPRIM_START_VERTEX	POCS
3DPRIM_VERTEX_COUNT	POCS
3DPRIM_XP0	POCS
3DPRIM_XP1	POCS
3DPRIM_XP2	POCS
IA_PRIMITIVES_COUNT	POCS
IA_VERTICES_COUNT	POCS
VS_INVOCATION_COUNT	POCS
CL_INVOCATION_COUNT	POCS
CL_PRIMITIVES_COUNT	POCS
MI_PREDICATE_DATA	POCS
MI_PREDICATE_RESULT	POCS
MI_PREDICATE_SRC0	POCS
MI_PREDICATE_SRC1	POCS
CSBEFSM	POCS
CSFLFLAG	POCS
CSFLFSM	POCS
CSFLTRK	POCS
CS_CONTEXT_STATUS1	POCS
CTX_RESTORE_ACK_0	POCS
CTX_RESTORE_ACK_1	POCS
FF_MODE	POCS
STATE_ACK	POCS
STATE_ACK_SLICE1	POCS
STATE_ACK_SLICE2	POCS
STATE_ACK_SLICE3	POCS
State_Ack_Register_Slice_5	POCS
State_Ack_Register_Slice4	POCS

POSH Functionality Controls

POSH functionality enabling and disabling is hierarchically controlled at various levels in the context execution flow.

- Context Granularity
- Batch Buffer Granularity
- 3DPRIMITIVE Granularity

POSH Control	Description
Context Granularity	<p>POSH feature can be enabled or disabled at context level by programming the "POSH Enable" field in CTX_SR_CTL register of the RCS. When POSH is disabled in CTX_SR_CTL register, RCS will not engage POSH.</p> <p>Usage model is one-time programming of "POSH Enable" field at context creation time.</p> <p>Dynamic enabling or disabling of POSH during context execution should be achieved through Batch Buffer and 3DPRIMITIVE granularity controls.</p>
Batch Buffer Granularity	<p>POSH Enable:</p> <p>"POSH Enable" field in MI_BATCH_BUFFER_START command indicates the possibility of encountering "POSH Start" batch buffer from the corresponding command sequence.</p> <p>POSH Start:</p> <p>Commands to be executed by the POCS must be programmed in a dedicated batch buffer and this batch buffer is indicated with a bit "POSH Start" in the MI_BATCH_BUFFER_START command. Once POCS encounters the batch buffer with "POSH Start" it executes all the command in the corresponding batch buffer and also the chained batch buffers from the corresponding buffer.</p> <p>RCS skips the MI_BATCH_BUFFER_START command with "POSH Start" set and goes on the following command.</p> <p>Programming Notes:</p> <p>POCS executes only the MI_BATCH_BUFFER_START commands programmed in the ring buffer with "POSH Enable" set and NOOPS (predicates) all the other commands in the ring buffer. POCS only parses/traverses the batch buffer with "POSH Enable" to check for any batch buffer programmed with "POSH Start" set.</p> <p>SW must set "POSH Enable" field in the MI_BATCH_BUFFER_START command programmed in ring buffer if the commands in the corresponding batch buffer or the chained batch buffers (includes Second Level and third level) has at least one batch buffer start command with "POSH Start" set (also implies 3DPRIMITIVE command for which POSH is enabled).</p>
3DPRIMITIVE Granularity	<p>"POSH Enable" field in the 3DPRIMITIVE command indicates the POSH pipe to create the visibility recording data and indicates Render pipe to use visibility recording data for the corresponding 3DPRIMITIVE command.</p>



POSH Interrupts

There are no interrupts generated by POSH pipe.

3D Pipeline Software Interface

This section covers the 3D pipeline software interface.

Synchronization of the 3D 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 of 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 in to 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 the 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

Argument	Bits	Restriction
		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	<p>Tile Cache Enabled Mode:</p> <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> ○ 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 <ul style="list-style-type: none"> ▪ 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 PIPECONTROL commands. SW must ensure not to issue any rendering commands between the two PIPECONTROL commands. <p>Unified Cache (Tile Cache Disabled):</p> <p>In unified cache mode of operation Color and Depth (Z) streams are cached in DC space of L2 along with Data Port stream.</p> <p>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.</p> <p>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.</p>

Arguments	Bit	Restrictions
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.
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.
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.

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 to independently.

As these engines are executing asynchronously:

- 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
<ul style="list-style-type: none"> • 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
<ul style="list-style-type: none"> • 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.

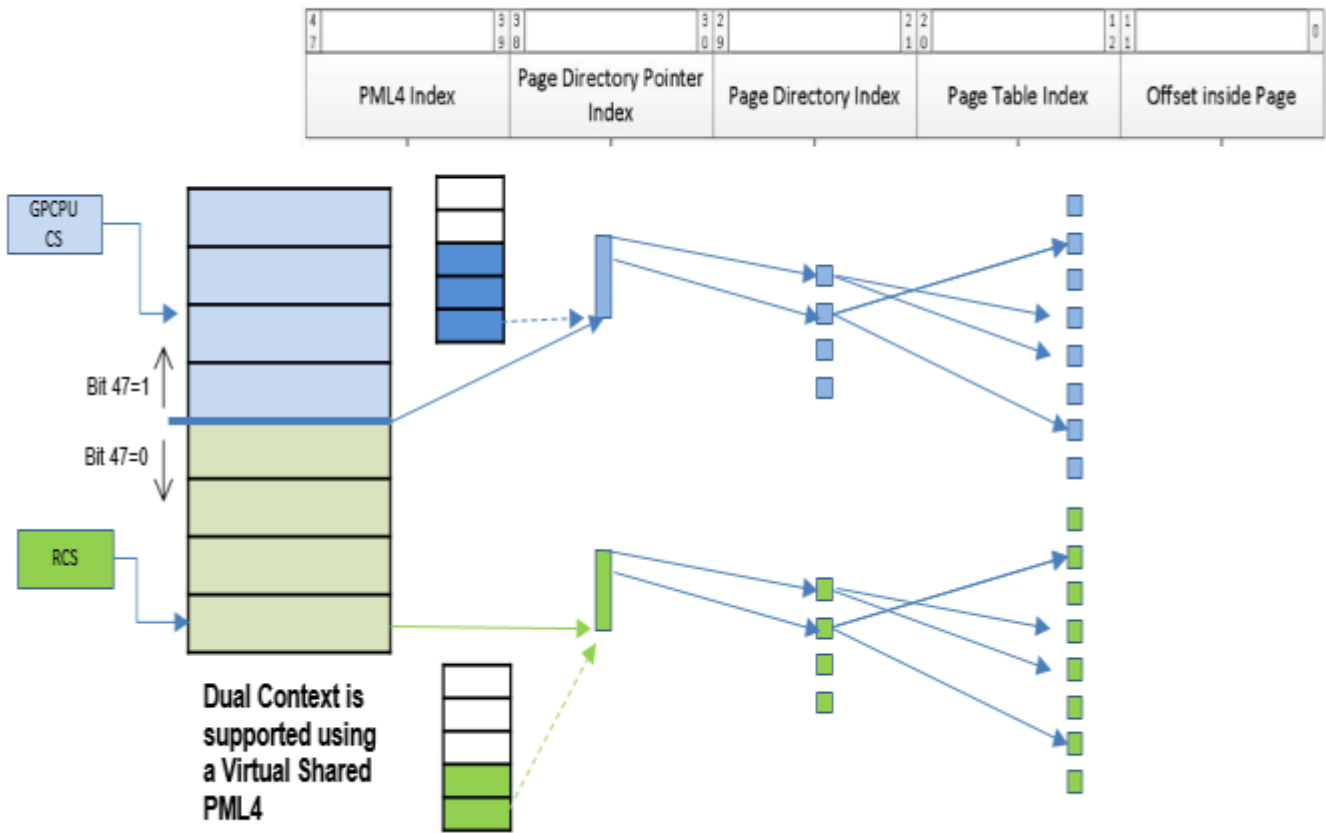
If the hardware is running multiple contexts and if a context needs to be reset (for e.g. because it is hung), then SW has to evict any other normally running context from the machine before it triggers Reset.

Address Space Considerations

Address Space Programming Note	
Context:	Address Space Constraints on Dual Context
<p>Dual context support is implemented by splitting the 48b address space into two halves:</p> <ul style="list-style-type: none">• RCS context occupies the address space corresponding to bit 47 = 0• GPGPU context occupies the address space corresponding to bit 47 = 1 <p>As a result of the above, a context can use the entire 48b address space when not running in dual context mode (matches legacy behavior). When running in dual context mode (i.e. Compute Engine Enable bit is set in Graphics Mode Register) contexts are limited to 47b address space.</p> <p>This model imposes several limitations for SW when running in dual context mode:</p> <ul style="list-style-type: none">• Dual context is supported only in Legacy Context mode with 64b VA• Applications shall be limited to 47b address space <p>As indicated earlier, dual context is supported by partitioning the address space (by effectively partitioning the PML4) as shown in figure below.</p>	

Address Space Programming Note

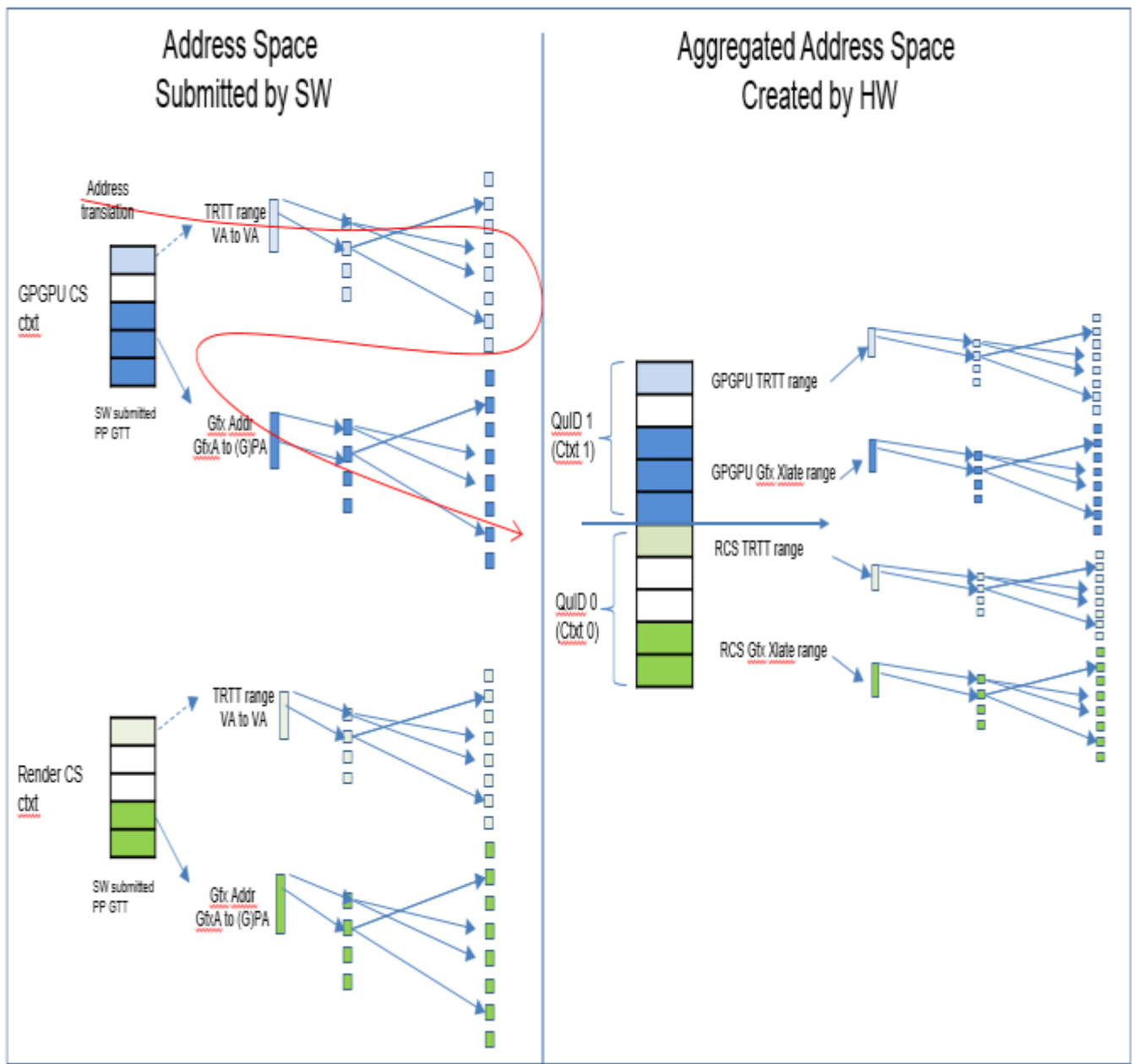
Context: Address Space Constraints on Dual Context



This shared address space needs to be configured to accommodate TR-TT. This is achieved by dropping the TR-TT address space to 46:44. The following figure depicts the TR-TT spaces in a dual context scenario.

Address Space Programming Note

Context: Address Space Constraints on Dual Context

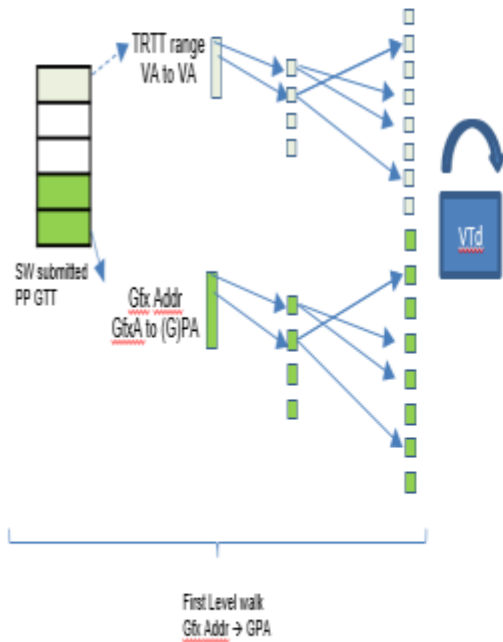


In a virtualized environment, each context can belong to a different Virtual Machine. The hardware allows for independent VT-d page table translation for each context.

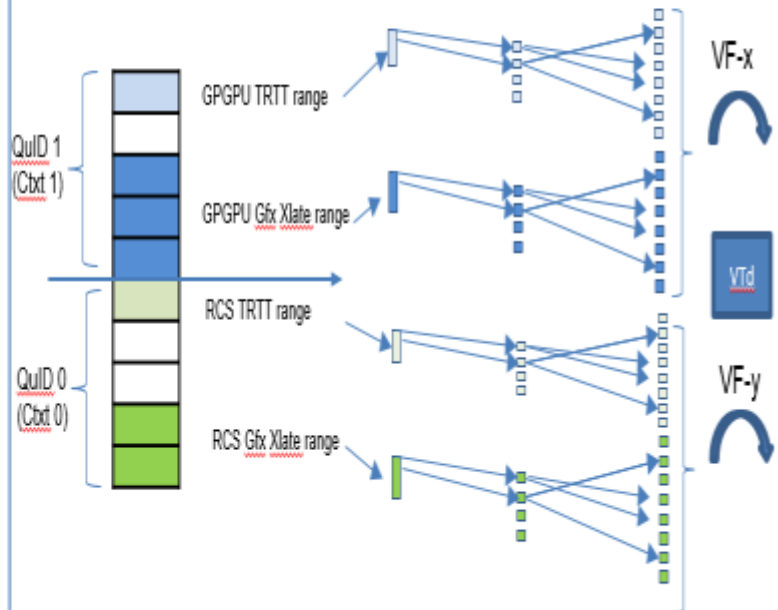
Address Space Programming Note

Context: Address Space Constraints on Dual Context

Address Translation with VT-d Legacy scenario



Address Translation with VT-d Dual Context scenario





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
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

Description	MMIO Offset/Command	Unit	# of DW
PDP0_UDW	0x1A274	CSEL	2
PDP0_LDW	0x1A270	CSEL	2
POSH_LRCA (DUMMY)	0x1A1B0	CSEL	2
CONTEXT_SCHEDULING_ATTRIBUTES (RESOURCE_MIN_MAX_PRIORITY)	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
NOOP		CSFE	1
MI_LOAD_REGISTER_IMM	0x1108_1065	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
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
NOOP		CSFE	1
MI_LOAD_REGISTER_IMM	0x1102_100F	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

Description	MMIO Offset/Command	Unit	# of DW
AUX_TABLE_BASE_ADDR_LOW	0x42C0	CSFE	2
AUX_TABLE_BASE_ADDR_HIGH	0x42C4	CSFE	2
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
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
NOOP		CSBE	8
NOOP		CSBE	12
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_1009	CS	1	00C5	CSBE
NOOP		CS	10	00D0	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_REPORT_PERF_COUNT
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
<ul style="list-style-type: none"> • MEDIA_VFE_STATE • MEDIA_CURBE_LOAD • MEDIA_INTERFACE_DESCRIPTOR_LOAD • MEDIA_OBJECT • MEDIA_OBJECT_* • MEDIA_OBJECT_GRPID • MEDIA_WALKER • GPGPU_WALKER



GPGPU-MEDIA State Commands
<ul style="list-style-type: none">MEDIA_STATE_FLUSH
Common Non-Pipeline State Commands
<ul style="list-style-type: none">STATE_BASE_ADDRESSGPGPU_CSR_BASE_ADDRESSPIPELINE_SELECTSTATE_SIP
<ul style="list-style-type: none">STATE_COMPUTE_MODE
Misc Commands
<ul style="list-style-type: none">PIPE_CONTROL3DSTATE_BINDING_TABLE_POOL_ALLOC
<ul style="list-style-type: none">L3_CONTROL

Engine State

This section describes the state specific to the 3D-Compute 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 the 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 <i>DataPort</i> 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
Indirect Object Base Address	MEDIA_OBJECT Indirect Data accessed by the CS unit
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 <u>except 'stateless' DataPort Read/Write requests</u> (e.g., RT accesses). See <i>DataPort</i> for a definition of the 'stateless' form of requests.

Base Address Used	Memory Accesses
None	Memory reads resulting from STATE_PREFETCH commands
None	Any physical memory access by the device
None	GTT-mapped accesses not included above (i.e., default)
None	Push Constants (depeding 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

POSH Power Context

The table below captures the data from CS power context save/restored by PM. Address offsets in this table are relative to the starting location of CS in the overall power context image managed by PM.

POCS Power Context Image

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
NOOP		POCS	1	0	CSFE
Load_Register_Immediate header	0x1100_0055	POCS	1	001	CSFE
GFX_MODE	0x1829C	POCS	2	0002	CSFE
GHWSP	0x18080	POCS	2	0004	CSFE
RC_PWRCTX_MAXCNT	0x18054	POCS	2	0008	CSFE
CTX_WA_PTR	0x18058	POCS	2	000A	CSFE
NOPID	0x18094	POCS	2	000C	CSFE
CMD_CCTL_0	0x180C4	POCS	2	000E	CSFE
PREEMPT_DLY	0x18214	POCS	2	0010	CSFE
CTXT_PREMP_DBG	0x18248	POCS	2	0012	CSFE
WAIT_FOR_RC6_EXIT	0x180CC	POCS	2	0014	CSFE
RCS_CTXID_PREEMPTION_HINT	0x184CC	POCS	2	0016	CSFE
CS_PREEMPTION_HINT_UDW	0x184C8	POCS	2	0018	CSFE
CS_PREEMPTION_HINT	0x184BC	POCS	2	001A	CSFE
MI_PREDICATE_RESULT_2	0x183BC	POCS	2	001C	CSFE
SEMA_WAIT_POLL	0x1824C	POCS	2	001E	CSFE
IDLEDELAY	0x1823C	POCS	2	0020	CSFE
RCS_FORCE_TO_NONPRIV_0_11	0x184D0	POCS	24	0022	CSFE
RCS_FORCE_TO_NONPRIV_12_15	0x18010	POCS	8	003A	CSFE
RCS_FORCE_TO_NONPRIV_16_19	0x181E0	POCS	8	0042	CSFE
EXECLIST_STATUS_REGISTER	0x18234	POCS	2	004A	CSFE
CXT_OFFSET	0x181AC	POCS	2	004E	CSFE
STOP_PARSER_CONTROL	0x18424	POCS	2	0050	CSFE
STOP_PARSER_HINT_ADDR	0x18428	POCS	4	0052	CSFE
NOOP		POCS	10	0056	CSFE
NOOP		POCS	1	0060	CSBE
Load_Register_Immediate header	0x1100_0011	POCS	1	0061	CSBE
FF_THREAD_MODE	0x180A0	POCS	2	0066	CSBE
VF PREEMPTION VERTEX HINT	0x16EB0	VFR	2	0070	CSBE
VF PREEMPTION INSTANCE HINT	0x16EB4	VFR	2	0072	CSBE

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
NOOP		POCS	10	0074	CSBE
NOOP		POCS	1	007E	CSBE
MI_BATCH_BUFFER_END		POCS	1	007F	CSBE

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
RS_PREEMPTION_HINT_UDW	0x24C4	CS	2	00E2	CSBE
RS_PREEMPTION_HINT	0x24C0	CS	2	00E4	CSBE
VF PREEMPTION VERTEX HINT	0x83B0	VF	2	00E8	CSBE
VF PREEMPTION INSTANCE HINT	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

TDL Power Context

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_004D	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
VSR_EMASK	E51C	TDL	2
SLM_BANK_HASH	E660	TDI	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_100D	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_1019	CSEL	1
	0x2244	CSEL	2
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
NOOP		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

Description	MMIO Offset/Command	Unit	# of DW
MI_LOAD_REGISTER_IMM	0x1108_1005	CSEL	1
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
NOOP		CSFE	1
MI_LOAD_REGISTER_IMM	0x1108_1065	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
NOOP		CSFE	1
MI_LOAD_REGISTER_IMM	0x1102_100F	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_INVALID	0x4414	CSFE	2

Description	MMIO Offset/Command	Unit	# of DW
AUX_TABLE_BASE_ADDR_LOW	0x4200	CSFE	2
AUX_TABLE_BASE_ADDR_HIGH	0x4204	CSFE	2
NOOP		CSFE	6
NOOP		CSBE	1
MI_LOAD_REGISTER_IMM	0x1108_10AD		
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
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
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

Description	MMIO Offset/Command	Unit	# of DW
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
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

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_PUSH_CONSTANT_ALLOC_GS		CSBE	2
3DSTATE_PUSH_CONSTANT_ALLOC_PS		CSBE	2
3DSTATE_BINDING_TABLE_POOL_ALLOC		CSBE	4
3DSTATE_GATHER_POOL_ALLOC		CSBE	4
3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC		CSBE	4
MI_RS_CONTROL		CSBE	1
3DSTATE_PTBR_TILE_PASS_INFO		CSBE	4
STATE_COMPUTE_MODE		CSBE	2
NOOP		CSBE	16
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
NOOP		SOL	3
3DSTATE_SO_DECL_LIST		SOL	259
NOOP		SOL	0
3DSTATE_INDEX_BUFFER		VF	5
3DSTATE_VERTEX_BUFFERS		VF	133
3DSTATE_VERTEX_ELEMENTS		VF	69
3DSTATE_VF_STATISTICS		VF	1
3DSTATE_VF		VF	2
3DSTATE_SGVS		VF	2
3DSTATE_VF_INSTANCING		VF	69
3DSTATE_VF_TOPOLOGY		VF	2
NOOP		VF	5

Description	MMIO Offset/Command	Unit	# of DW
MI_LOAD_REGISTER_IMM	0x1100_10C7	VF	1
INSTANCE CNT	08300 - 08384h	VF	68
INSTANCE INDX	08400 - 08484h	VF	68
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
INDEX_OPCODE_DATA00	08490h	VF	2
INDEX_OPCODE_DATA01	08494h	VF	2
INDEX_OPCODE_DATA10	08498h	VF	2
INDEX_OPCODE_DATA11	0849Ch	VF	2
TOKPROC_CULL_COUNT0	084A0h	VF	2
TOKPROC_CULL_COUNT1	084A4h	VF	2
TOKPROC_PID_COUNT0	084A8h	VF	2
TOKPROC_PID_COUNT1	084ACh	VF	2
TOKPROC_CULL_VERTEX	084B0h	VF	2
TOKPROC_PID_OBJECT	084B4h	VF	2
TOKPROC_DUMMY_OBJECT	084B8h	VF	2
TOKPROC_CL_PTR	084BCh	VF	2
TOKPROC_CL_MISC	084C0h	VF	2
TOKPROC_STG1_DATA	084C4h	VF	2
TOKPROC_STG1_VERTEX_COUNT	084C8h	VF	2
TOKPROC_STG1_OBJECT_COUNT	084CCh	VF	2
TOKPROC_STG1_VALID	084D0h	VF	2
TOKPROC_STG0_INSTANCE_COUNT	084D4h	VF	2
TOKPROC_STG0_VERTEX_COUNT	084D8h	VF	2
TOKPROC_STG0_COUNT	084DCh	VF	2
TOKPROC_STG0_VALID	084E0h	VF	2
TOKIN_DATA0	084F0h	VF	2
TOKIN_DATA1	084F4h	VF	2
TOKIN_DATA2	084F8h	VF	2
TOKIN_DATA3	084FCh	VF	2
NOOP		VF	7
3DSTATE_VF_COMPONENT_PACKING		VF	5

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_VF_SGVS_2		VF	3
3DSTATE_PTBR_TILE_SELECT		VF	2
NOOP		VF	2
NOOP		VF	4
OAR Cotext 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
L3RCSORSVD1	0xB168	LBCF	2
3DSTATE_CONSTANT_VS_Committed		SVG	11
3DSTATE_CONSTANT_HS_Committed		SVG	11
3DSTATE_CONSTANT_DS_Committed		SVG	11
3DSTATE_CONSTANT_GS_Committed		SVG	11
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_SF		SVG	4
3DSTATE_SCISSOR_STATE_POINTERS		SVG	2
3DSTATE_VIEWPORT_STATE_POINTERS_CL_SF		SVG	2
3DSTATE_RASTER		SVG	5
3DSTATE_WM_HZ_OP		SVG	5
3DSTATE_MULTISAMPLE		SVG	2
3DSTATE_HS		SVG	9

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_BINDING_TABLE_POINTERS_HS		SVG	2
3DSTATE_SAMPLER_STATE_POINTERS_HS		SVG	2
3DSTATE_URB_ALLOC_HS		SVG	3
3DSTATE_TE		SVG	5
3DSTATE_DS		SVG	11
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
NOOP		SVG	1
3DSTATE_CONSTANT_VS_NonComitted		SVG	11
3DSTATE_CONSTANT_HS_NonComitted		SVG	11
3DSTATE_CONSTANT_DS_NonComitted		SVG	11
3DSTATE_CONSTANT_GS_NonComitted		SVG	11
3DSTATE_DRAW_RECTANGULAR		SVG	4
MI_LOAD_REGISTER_IMM	0x1100_1007	SVG	1
FF_PERF_REG	0x6b1c	SVG	2
DEREF_CTRL	0x6000	SVG	2
FF_MODE2	0x6604	SVG	2
CULLBIT1	0x6100	SVG	2
NOOP		SVG	8
NOOP		SVG	1
NOOP		SVG	3
NOOP		SVG	2
NOOP		SVG	1
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

Description	MMIO Offset/Command	Unit	# of DW
3DSTATE_DEPTH_BUFFER		SVL	8
3DSTATE_HIER_DEPTH_BUFFER		SVL	5
3DSTATE_STENCIL_BUFFER		SVL	8
3DSTATE_CLEAR_PARAMS		SVL	3
3DSTATE_CPS_POINTERS		SVL	2
3DSTATE_DEPTH_CNTL_BUFFER		SVL	5
3DSTATE_DEPTH_BOUNDS		SVL	4
3DSTATE_AMFS_TEXTURE_POINTERS		SVL	11
3DSTATE_SBE		SVL	6
3DSTATE_SBE_SWIZ		SVL	11
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
NOOP		SVL	1
3DSTATE_CONSTANT_PS_NonComitted		SVL	11
3DSTATE_3D_MODE		SVL	2
3DSTATE_SAMPLE_PATTERN		SVL	9
3DSTATE_SUBSLICE_HASH_TABLE		SVL	14
NOOP		SVL	1
NOOP		SVL	12
MI_LOAD_REGISTER_IMM	0x1100_101F	SVL	1
Cache_Mode_0	0x7000	SVL	2
Cache_Mode_1	0x7004	SVL	2
GT_MODE	0x7008	SVL	2
FBC_RT_BASE_ADDR_REGISTER	0x7020	SVL	2
FBC_RT_BASE_ADDR_REGISTER_UPPER	0x7024	SVL	2
OA_CULL	0x7030	SVL	2
PSS_MODE	0x7038	SVL	2
PSS_MODE2	0x703C	SVL	2
NOOP		SVL	8
NOOP		SVL	14

Description	MMIO Offset/Command	Unit	# of DW
NOOP		SVL	2
NOOP		WM	1
MI_LOAD_REGISTER_IMM	0x1100_1007	WM	1
WMHWCLRVAL	0x5524	WM	2
3DSTATE_POLY_STIPPLE_PATTERN		WM	33
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	11
3DSTATE_MONOFILTER_SIZE		SC	2
3DSTATE_CHROMA_KEY		SC	16
NOOP		SC	14
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 register's description and related feature for more information on each individual bit.

The registers detailed in this chapter are used across the family of 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 3D-Compute 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
L3CNTLREG - L3 Control Register



Register
CACHE_MODE_SS - Cache Mode Subslice Register

Pipelines Statistics Counter Registers

These registers keep continuous count of statistics regarding the 3D 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.

Register
CS_CTX_TIMESTAMP - CS Context Timestamp Count

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

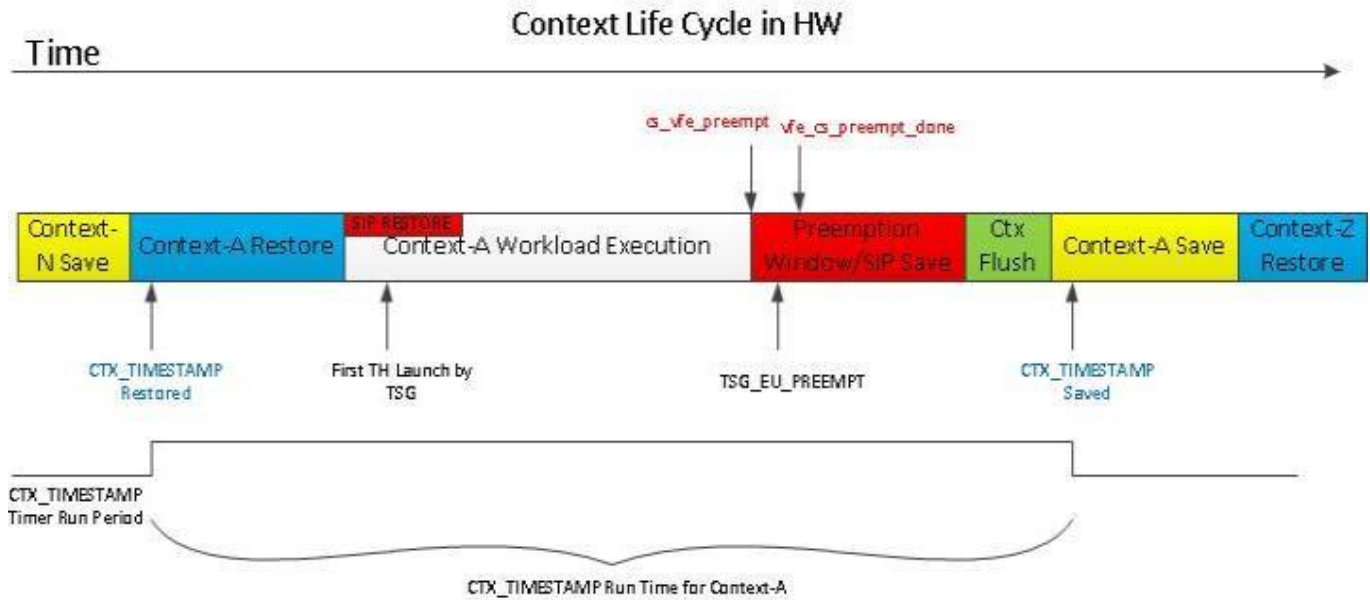


Fig: CS_CTX_TIMESTAMP functionality during context execution

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

AUTO_DRAW Registers

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 both the 3D and Media pipelines. This state is comprised of CS FF unit state, non-pipelined global state (EU, etc.), and Sampler shared-function state.

- 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 (i.e., 3DPRIMITIVE or MEDIA_OBJECT).

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.

Media Pipeline-Specific State-Setting Commands

Media Pipeline-Specific State-Setting Commands

The following command is used to set state specific to the Media pipeline:

- MEDIA_STATE_POINTERS

The state variables associated with this command must be set appropriately prior to issuing MEDIA_OBJECT.

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.

MEDIA_OBJECT

Before issuing a MEDIA_OBJECT command, all state (with the exception of 3D-pipeline-specific state) needs to be valid. Therefore, the commands used to set this state need to have been issued at some point prior to the issue of MEDIA_OBJECT.

3D Pipeline Stages

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

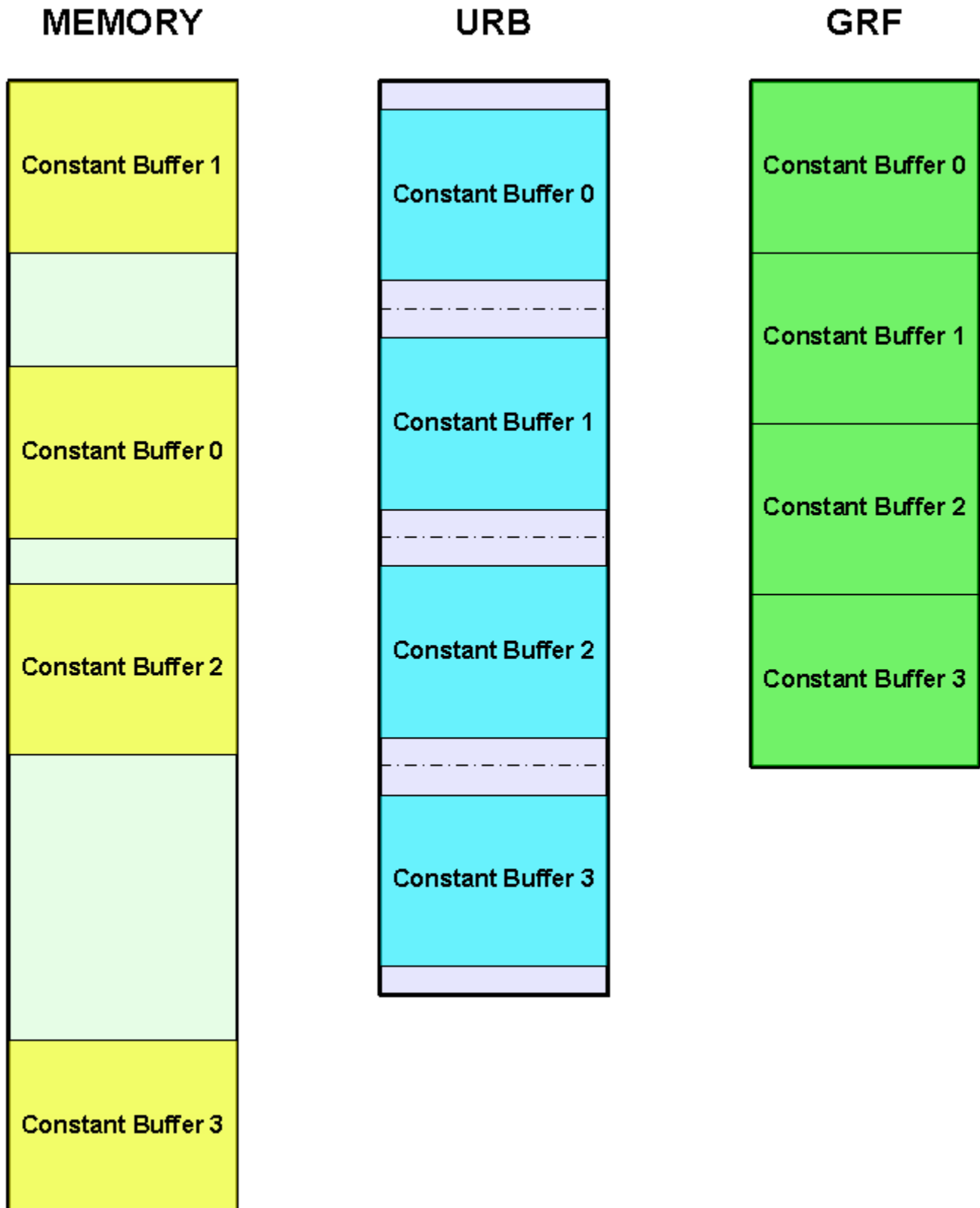
Pipeline Stage	Functions Performed
Command Stream (CS)	The Command Stream stage is responsible for managing the 3D pipeline and passing commands down the pipeline. In addition, the CS unit reads "constant data" from memory buffers and places it in the URB. Note that the CS stage is shared between the 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
<p>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.</p> <p>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.</p> <p>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.</p>	

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.

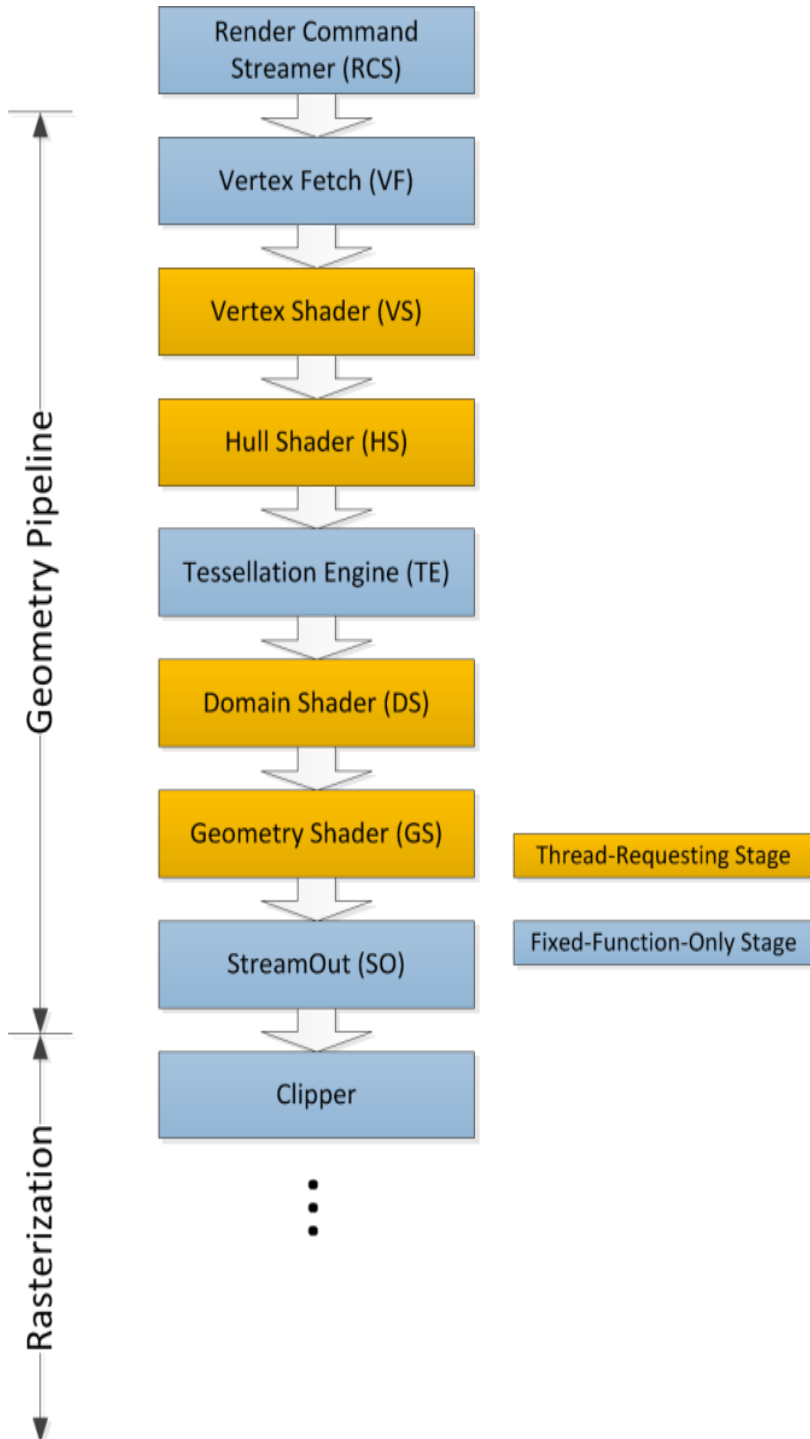
Software must guarantee that there is enough space in the push constant buffer in the URB to hold one constant buffer from memory. This includes any buffering to write the 512b aligned requests from memory into the URB. Because the L3\$ only supports writes from memory in 512b chunks, the URB may have some bubbles between each constant buffer fetch.

3DSTATE_3D_MODE

3D Pipeline Geometry

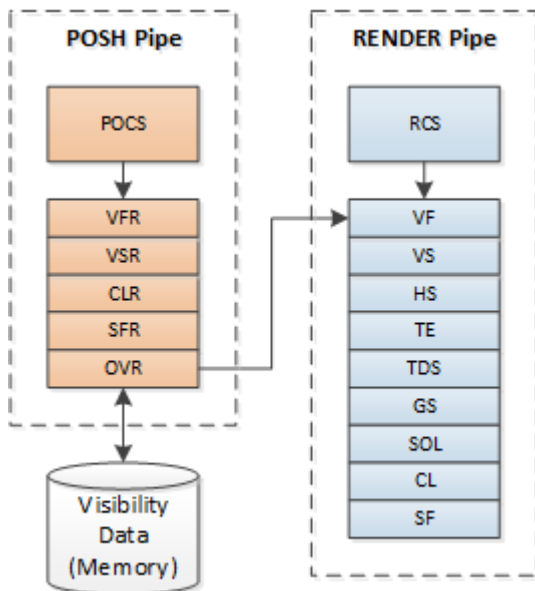
Block Diagram

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



POSH Pipeline Overview

The **Position-Only Shading (POSH) pipeline** (aka "Cull Pipe" or "Record Pipe") is utilized to improve 3D rendering performance by removing culled objects from the Render pipeline workload. The POSH pipeline pre-processes geometry objects using simple "position-only" vertex input and vertex shaders. These objects are then subjected to clipper/setup cull tests. The results of these cull tests are then stored (compressed) as streams of "visibility tokens" in memory. Later, when the same geometry work is submitted to the Render pipe, the VF stage of the Render pipe will receive the pre-recorded visibility tokens and use those tokens to skip over culled objects and only process the non-culled objects. The POSH pipe is designed to run ahead of the Render pipe by buffering visibility data for render passes and possibly entire frames before being consumed by the Render pipe.



POSH Pipeline Work Submission

Work is performed on the POSH pipeline by submitting command streams to the POSH CS (POCS) unit which operates similarly to the Render CS (RCS) unit. Refer to *Command Stream Programming* for POCS programming details.

Geometry & Setup Stages of POSH Pipeline

The POSH pipeline contains POSH-specific versions of a subset of the Render pipe stages:

- VFR (POSH VF)
- VSR (POSH VS)
- CLR (POSH CL)
- SFR (POSH SF)



Note that the POSH pipeline does not contain HS, DS, GS or StreamOut stages and therefore does not support those functions. Work submitted to the POSH pipeline shall not contain state commands for those stages do not attempt to enable those functions.

These POSH stages are programmed in a similar manner as the corresponding Render stages. When stage-related state commands are submitted to the POSH pipeline, the corresponding stages in the POSH pipeline are programmed. POSH/Render pipeline programming differences are described in the state command definitions.

OVR Stage of POSH Pipeline

An Object Visibility Recording (OVR) stage is located at the end of the POSH pipe. It is used to compress and store visibility token streams in memory, as well as reading those streams during rendering and passing the tokens to the VF stage. Refer to the *Render Engine Command Streamer BXML* for programming details.

URB Programming when POSH Enabled

When the POSH pipeline is enabled, a URB allocation for the VSR stage is required. This allocation is programmed via execution of `3DSTATE_URB*_VS` commands in the POSH pipeline. Software shall be required to manage this allocation, taking into account the synchronous operation of the RCS and POCS workloads. This programming may require explicit synchronization between the pipelines, e.g., when Render vs. POSH URB allocation boundaries are changed.

When the POSH pipeline is enabled, a URB allocation for the POSH pipeline Push Constants may be defined. Refer to the relevant Push Constant URB commands for details on how this allocation is defined and used.

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** (FPMMode, 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
Per-Thread Scratch Space	Y	Y	Y	Y
Scratch Space Base Pointer	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 **Per-Thread Scratch Space** state variable specifies the amount of scratch memory required by each active thread of a stage. The value is specified as a 4-bit power of two (in excess of 10) bytes, where programmed values in the valid range [0,11] specify scratch space requirements in the range [1KB, 2MB].

When a thread becomes "active" it is allocated a portion of scratch space, sized according to **PerThreadScratchSpace**. The starting location of each thread's scratch space allocation, **ScratchSpaceOffset**, is passed in the thread payload in R0.5[31:10] and is specified as a 1KB-granular offset from the **GeneralStateBaseAddress**. The computation of **ScratchSpaceOffset** includes the starting address of the stage's scratch space allocation, as programmed by **ScratchSpaceBasePointer**. The maximum number of active threads for a stage is specified by the **MaximumNumberOfThreads** state. SW shall abide by the scratch space restrictions included in the description of **MaximumNumberOfThreads**.

This value is also included within thread payloads in R0.3[3:0]. If a thread can potentially send any "A32 Stateless" messages to the DataPort shared function, that thread shall ensure that it passes along the **PerThreadScratchSpace** value passed in the thread payload.

The state command specifies starting offset of the scratch memory region allocated to a stage (**Scratch Space Base Pointer**). It is specified as a 22-bit, 1KB-aligned offset from the **GeneralStateBaseAddress**.

Each thread requested by the FF stage will be allocated its exclusive portion of this space, with the per-thread allocation size specified by **Per-Thread Scratch Space**. The computed offset of the thread-specific portion is passed in the thread payload as **Scratch Space Offset**. If the thread needs to access this scratch space, it shall utilize "stateless" DataPort read/write message, where the DataPort will cause the **General State Base Address** to be added to the specific scratch space offset passed in the message header.

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

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.

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*.

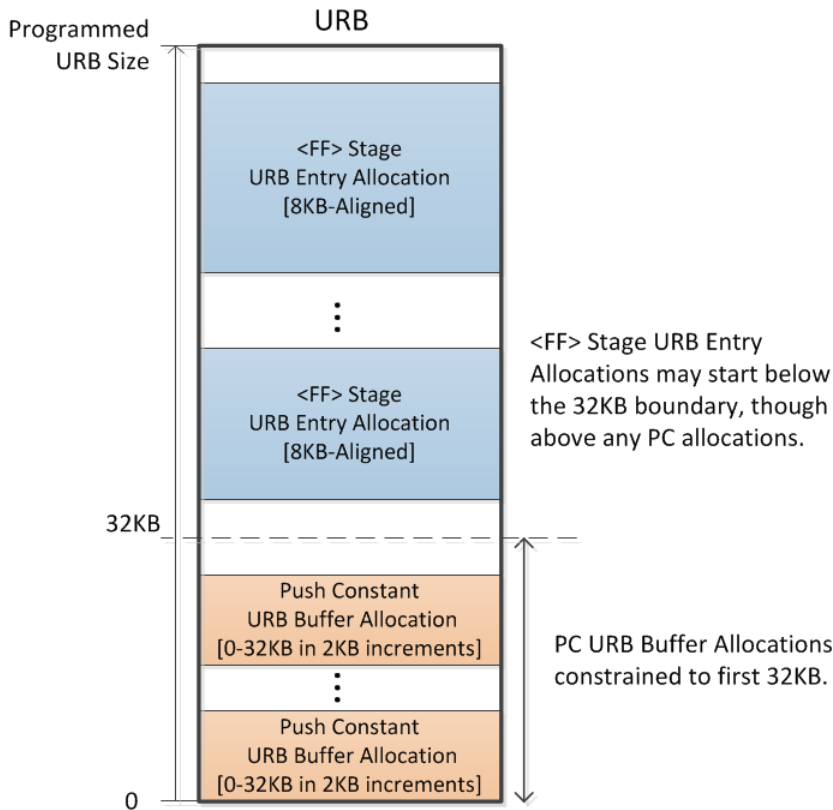
Software can place URB Entry allocations following any PC URB Buffer allocations. Software shall define allocations for all the relevant Geometry FFs (VS, HS, DS, GS), though a subset of these allocations can be "null" allocations that do not consume URB space. The VS stage always requires a non-null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

When POSH is Enabled (via CTXT_SR_CTL), an additional 32KB block of URB is allocated for POCS pipeline Push Constants.

This block is located immediately after the RCS Push Constant URB Buffer Allocation.

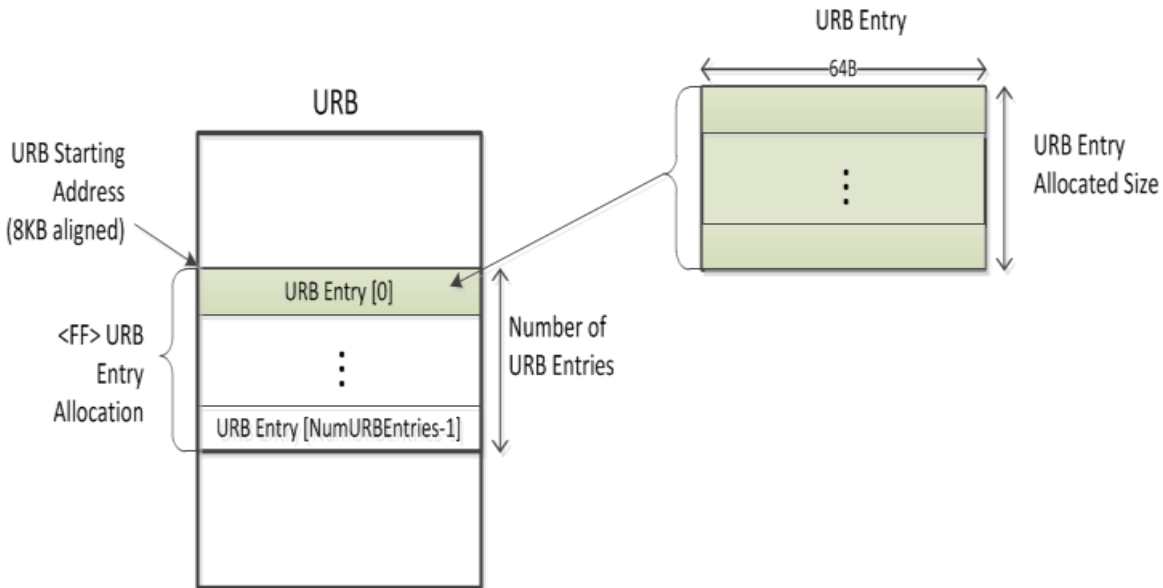
When enabled, the size of the Push Constant URB allocation mentioned in the URB programming information (below) will increase to 64KB total (vs. the 32KB size shown)

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.

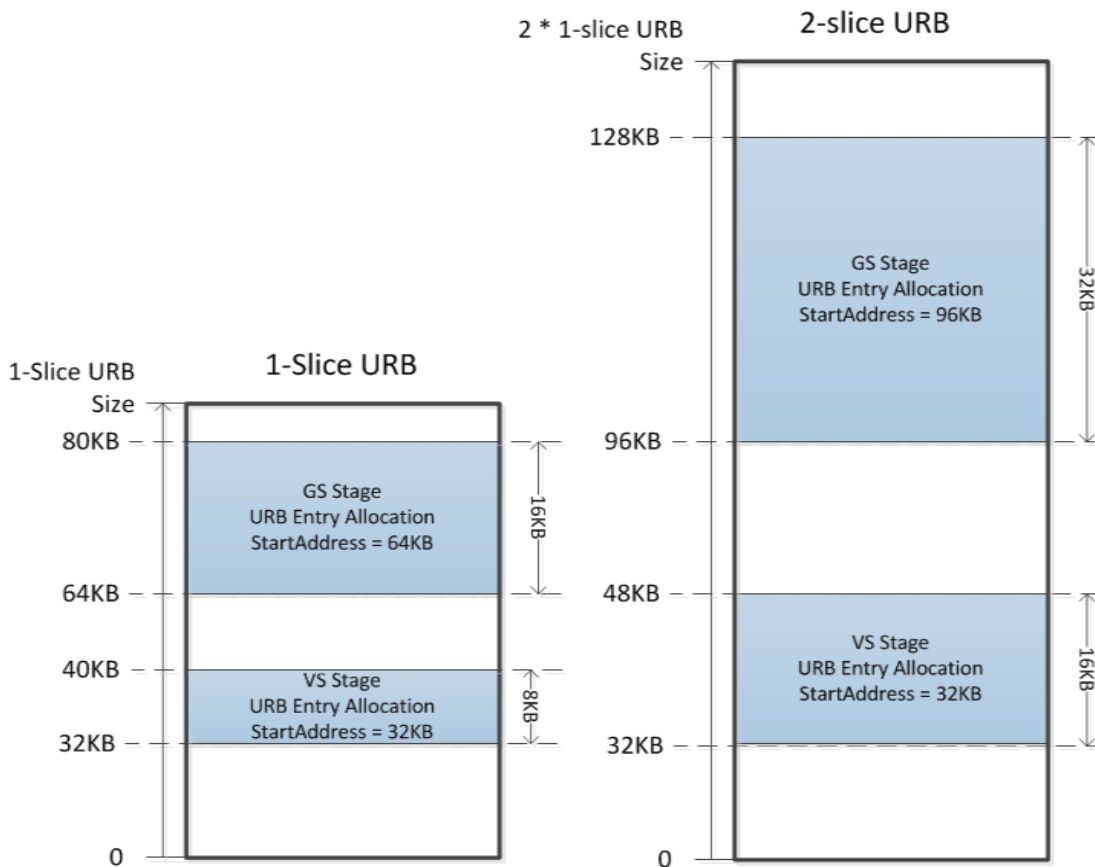
URB Entry and Entry Allocation



Multiple-Slice Implications

The FF URB allocations are programmed based on the URB size for a single slice. If the configuration includes multiple slices, the HW will automatically adjust the **URBStartAddress** and **NumberOfURBEntries** states according to the number of slices. The **URBEntryAllocationSize** states are not affected, nor are the PC URB Buffer allocations affected. The **NumberOfURBEntries** states are simply multiplied by the number of slices. The **URBStartAddress** states are scaled by the number of slices after being first decreased by 32KB, with a 32KB offset added back in after scaling. This scales the allocation start addresses relative to the 32KB boundary versus the start of the URB.

The following diagram provides an example of how this scaling would be applied in a 2-slice configuration.



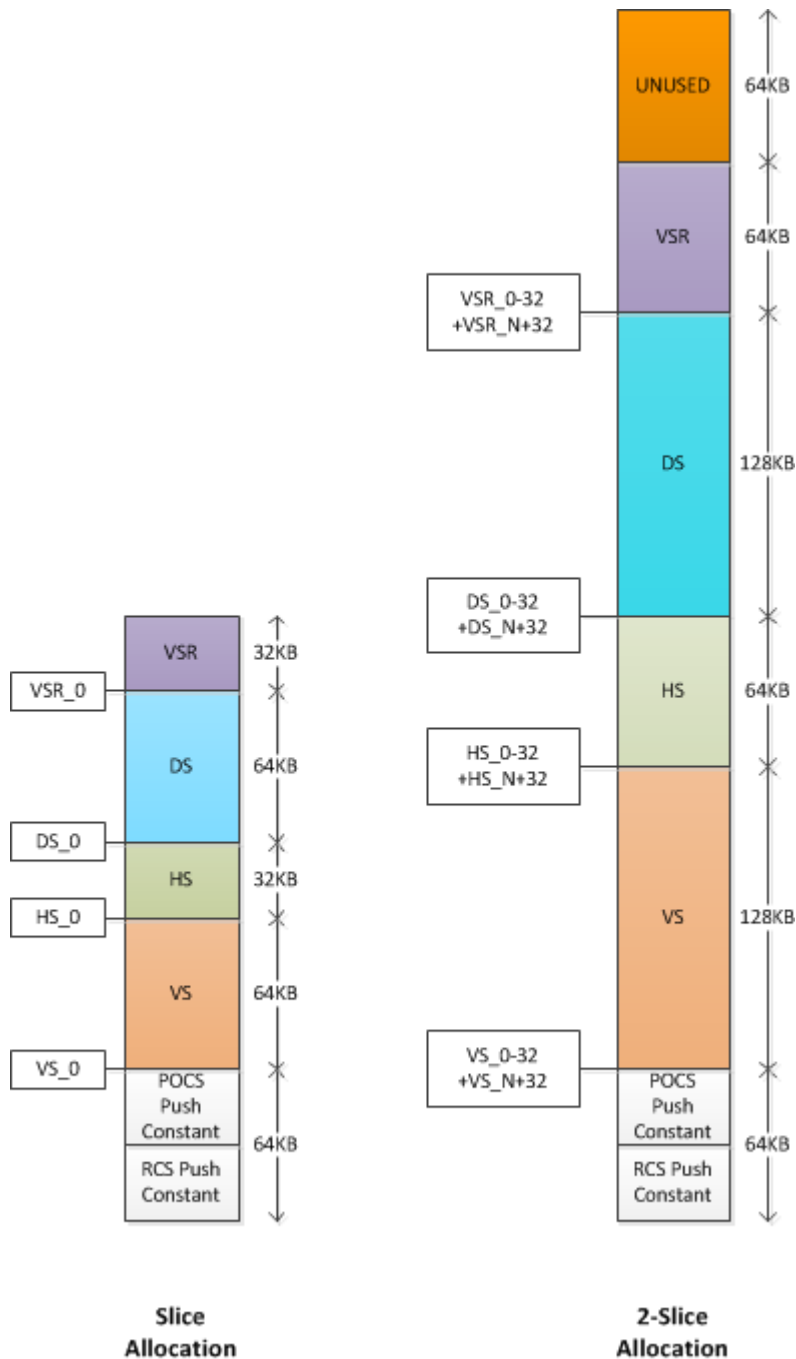
Enhanced URB Allocation Overview

An "enhanced", alternative FF URB allocation scheme is available and offers more efficient URB allocation for multiple-slice configurations when Push Constant URB allocations are required. This scheme uses the 3DSTATE_URB_ALLOC_VS, 3DSTATE_URB_ALLOC_HS, 3DSTATE_URB_ALLOC_DS and 3DSTATE_URB_ALLOC_GS commands. The states programmed by these commands are similar to the legacy 3DSTATE_URB_VS, 3DSTATE_URB_HS, 3DSTATE_URB_DS and 3DSTATE_URB_GS commands, though the new commands support separate URB Starting Address and Number Of URB Entries states for "Slice0" and "SliceN" (slices beyond Slice0). HW will use both sets of states to scale the URB allocation up to the number of enabled slices, versus the legacy commands that only used the single set of states for Slice0.

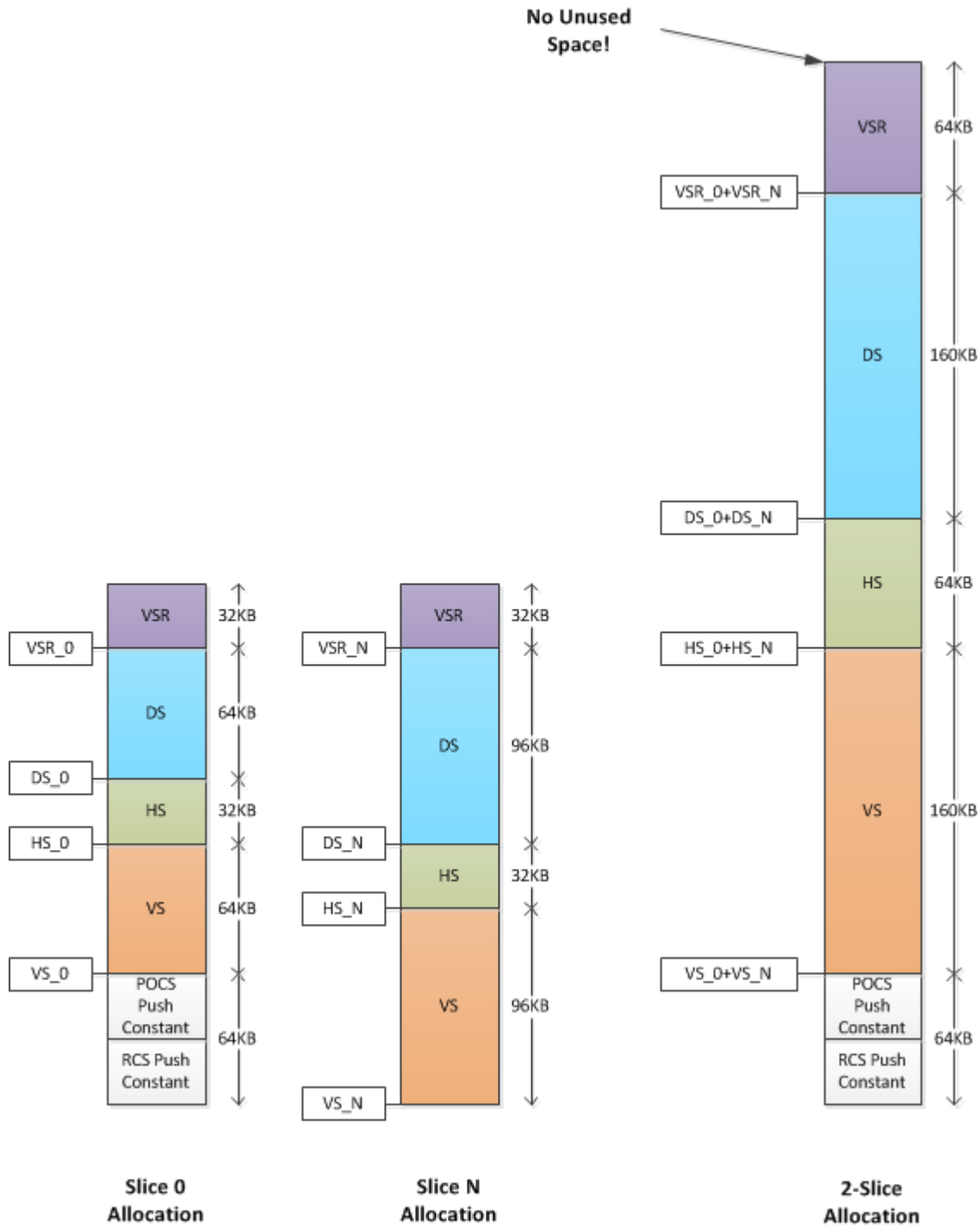
The enhanced scheme & commands are available in addition to the legacy scheme & commands. However, within a context, SW shall only utilize one scheme.

The benefit of the enhanced scheme is apparent for multiple-slice configurations when Push Constant URB allocations are required. The legacy scheme only supplied a single set of states for one slice. The size of the FF allocations are therefore impacted by the size of the Push Constant allocations. However, in a multiple-slice configuration, there is only one set of Push Constant allocation. The legacy scheme was unable to take advantage of that fact, resulting in *unused URB storage* at the end of the URB space equal

in size to N-1 times the size of the Push Constant allocations (where N is the number of enabled slices). The following example diagram illustrates this condition associated with the legacy scheme:



The enhanced scheme allows a separate, second set of states for "SliceN" (slices beyond Slice0) that can be used to specify allocations that are not impacted by Push Constant URB allocations (as these allocations do not exist in slices beyond Slice0). Therefore, the entire URB can be allocated with no wasted space. The following example diagram illustrates how the use of the enhanced scheme leads to more efficient FF URB allocation:



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
<ul style="list-style-type: none"> 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
<ul style="list-style-type: none"> 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 (1 st 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 (SIMDn _{xm} with m = 1) or multiple program flows (SIMDn _{xm} with m > 1). See CR0 description in <i>ISA Execution Environment</i> .	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 handling logic in the EU. (See ISA.)	All FFs spawning threads
Sampler Count	This is a <u>hint</u> 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 <u>hint</u> 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

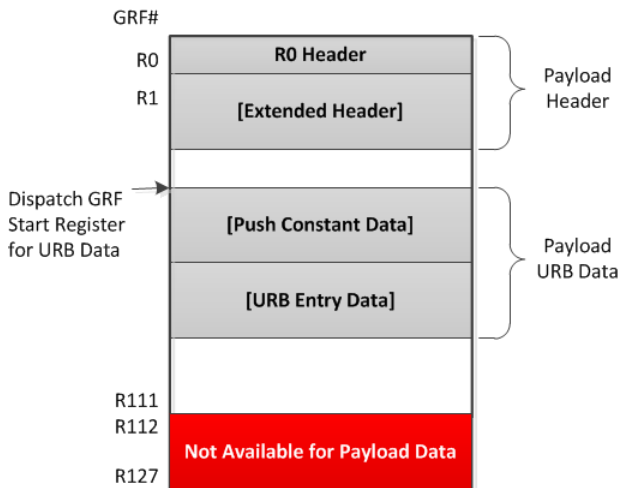
Thread Payload Overview

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 follow 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 <i>not</i> uniquely identify the thread over any significant	All FFs spawning threads

Fixed Payload Header Field (non-FF-specific)	Description	FFs
	<p>period.</p> <p>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.</p>	
Binding Table Pointer	<p>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 <i>Shared Functions</i> for a description of a Binding Table.</p>	All FFs spawning threads
Sampler State Pointer	<p>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 <i>Shared Functions</i> for a description of a Sampler State Table.</p>	All FFs spawning threads which sample (VS, GS, WM)
Per Thread Scratch Space	<p>This field specifies the amount of scratch space allocated to each thread spawned by the FF unit.</p> <p>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.</p>	All FFs spawning threads
Handle ID <n>	<p>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.</p> <p>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.</p>	VS, GS, CLIP, SF
URB Return Handle <n>	<p>This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry.</p> <p>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.</p>	VS, GS, CLIP, SF
Primitive Topology Type	<p>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 led to the spawning of the thread.</p> <p>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</p>	GS, CLIP, SF, WM

Fixed Payload Header Field (non-FF-specific)	Description	FFs
	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).	

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 Offset	R0.5[31:10]
Per Thread Scratch Space	R0.3[3:0]
Sampler State Pointer	R0.3[31:5]

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 Offset

This field specifies the starting offset of the 1KB-aligned scratch space region allocated to this particular thread. See the definition of **ScratchSpaceBaseAddressLow**, which specifies the starting offset for scratch space region allocated to the FF stage.

Per Thread Scratch Space

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

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	<p>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.</p> <p>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.</p> <p>Specifying a vertex data source extending beyond the end of a vertex entry is UNDEFINED.</p>	VS, GS
Vertex URB Entry Read Length	<p>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.</p> <p>A zero value is INVALID (at very least one 256-bit unit must be read).</p> <p>Specifying a vertex data source extending beyond the end of a VUE is UNDEFINED.</p>	

Programming Restrictions: (others may already be mentioned)

- The maximum size payload for any thread is limited by the number of GRF registers available to the thread, as determined by $\min(128, 16 * \text{GRF Register Block Count})$. 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.

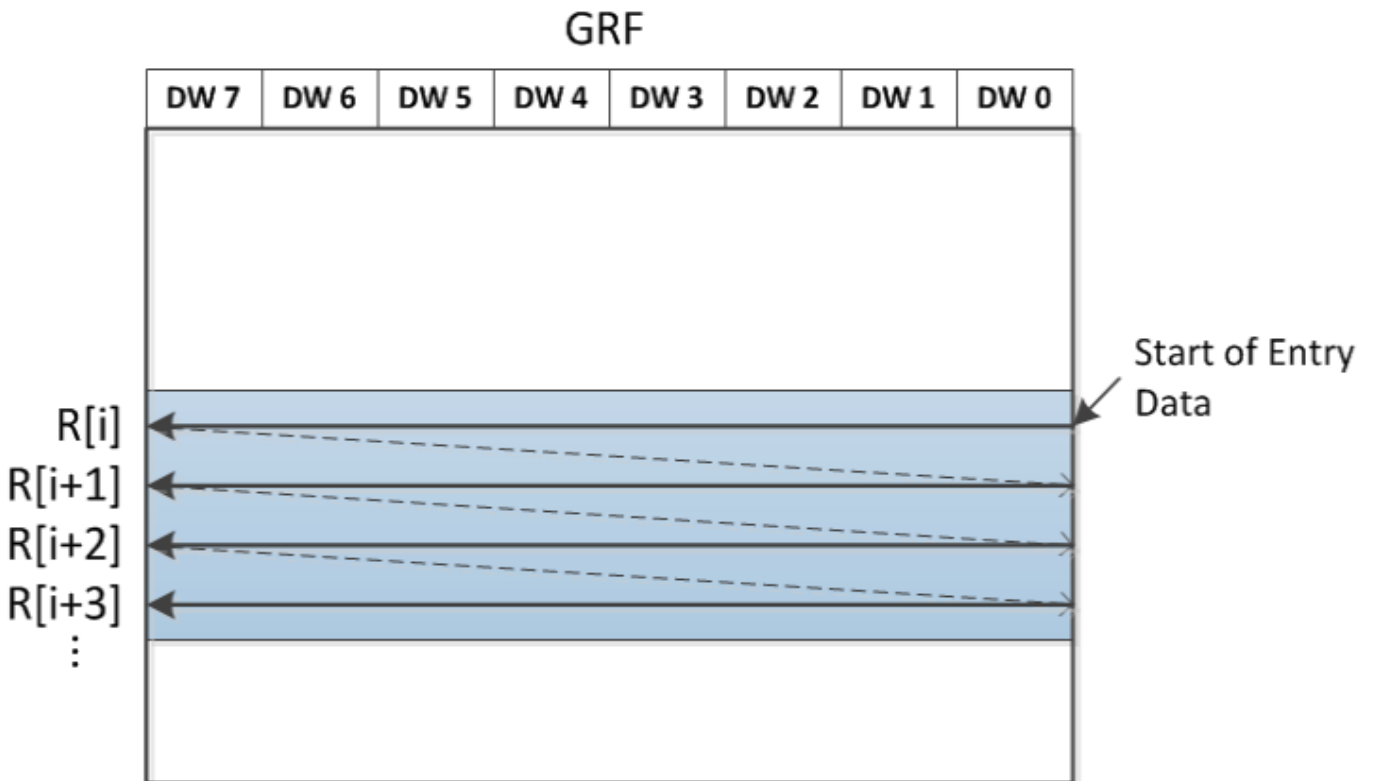
- For any type of URB Entry reads:
 - Specifying a source region (via Read Offset, Read Length) that goes past the end of the URB Entry allocation is illegal.
 - The allocated size of Vertex/Primitive URB Entries is determined by the **URB Entry Allocation Size** value provided in the pipeline state 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).

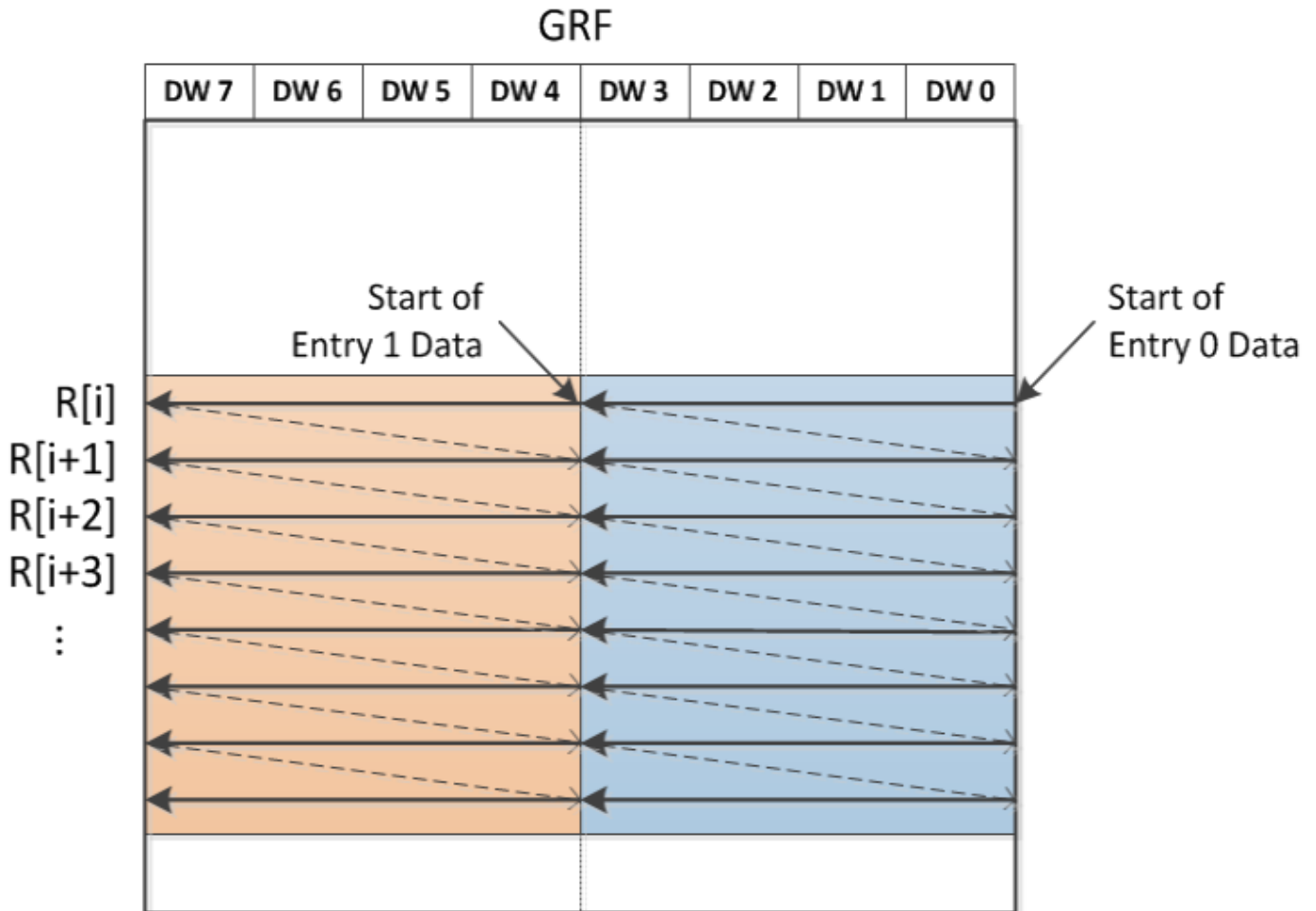
Linear

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.



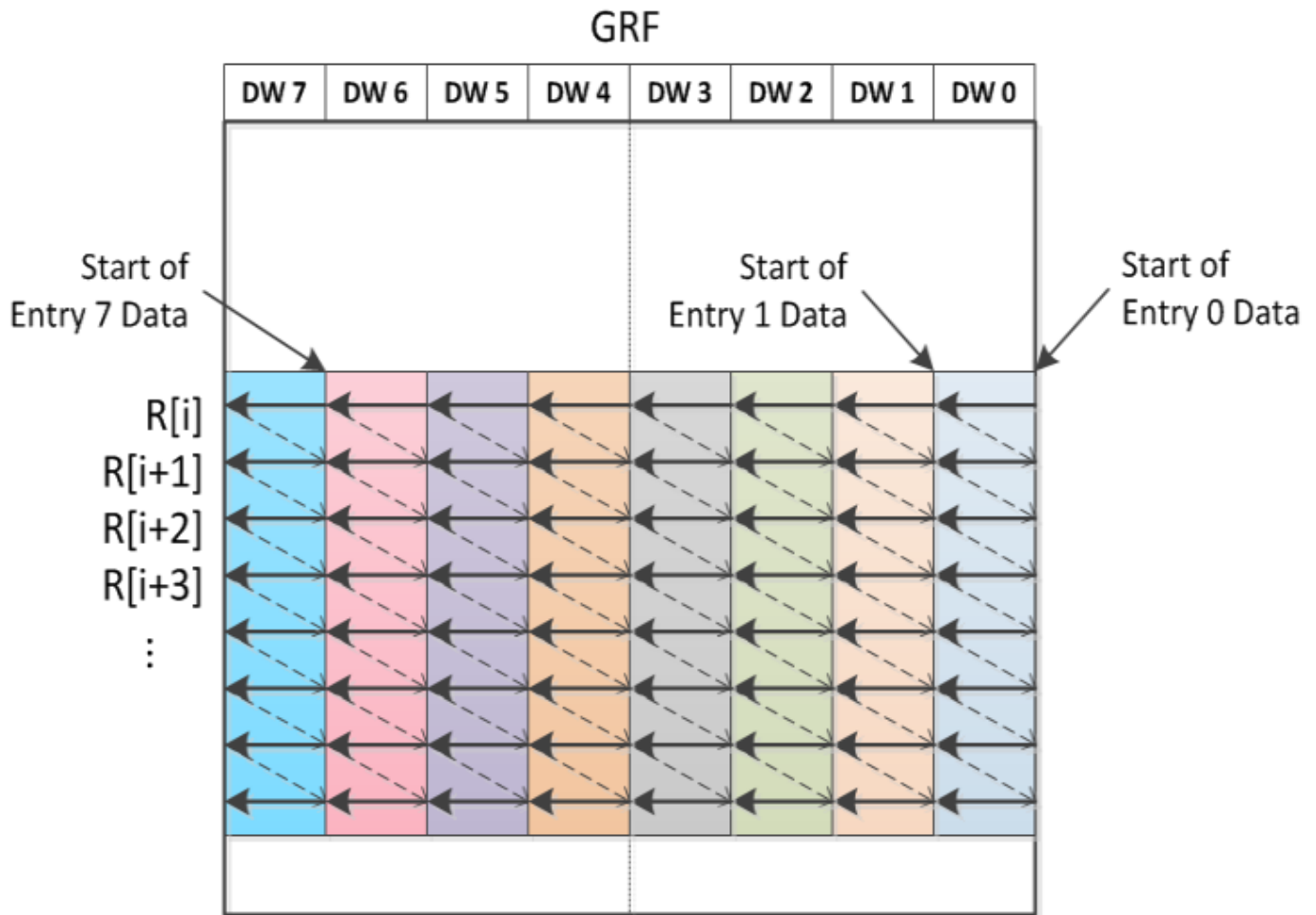
SIMD4x2 Interleaved

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

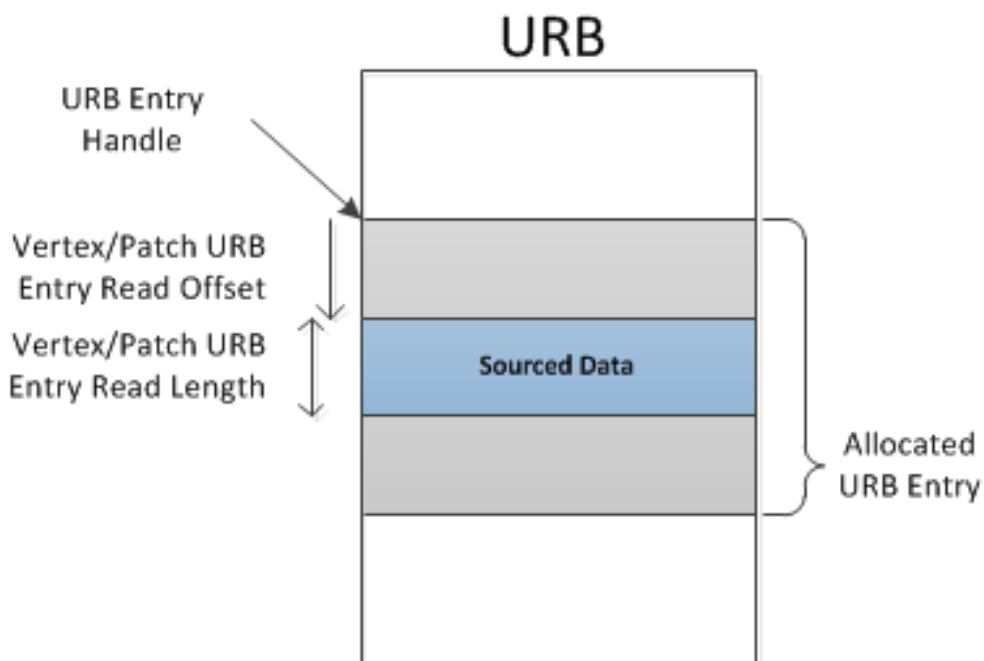
This section of the Payload URB Data is used to pass Push Constant data to the thread kernel. Software can define up to 4 Push Constant Memory Buffer regions for each Geometry FF stage that requests thread dispatches, after which the contents of those memory regions are automatically included in each 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 are 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:

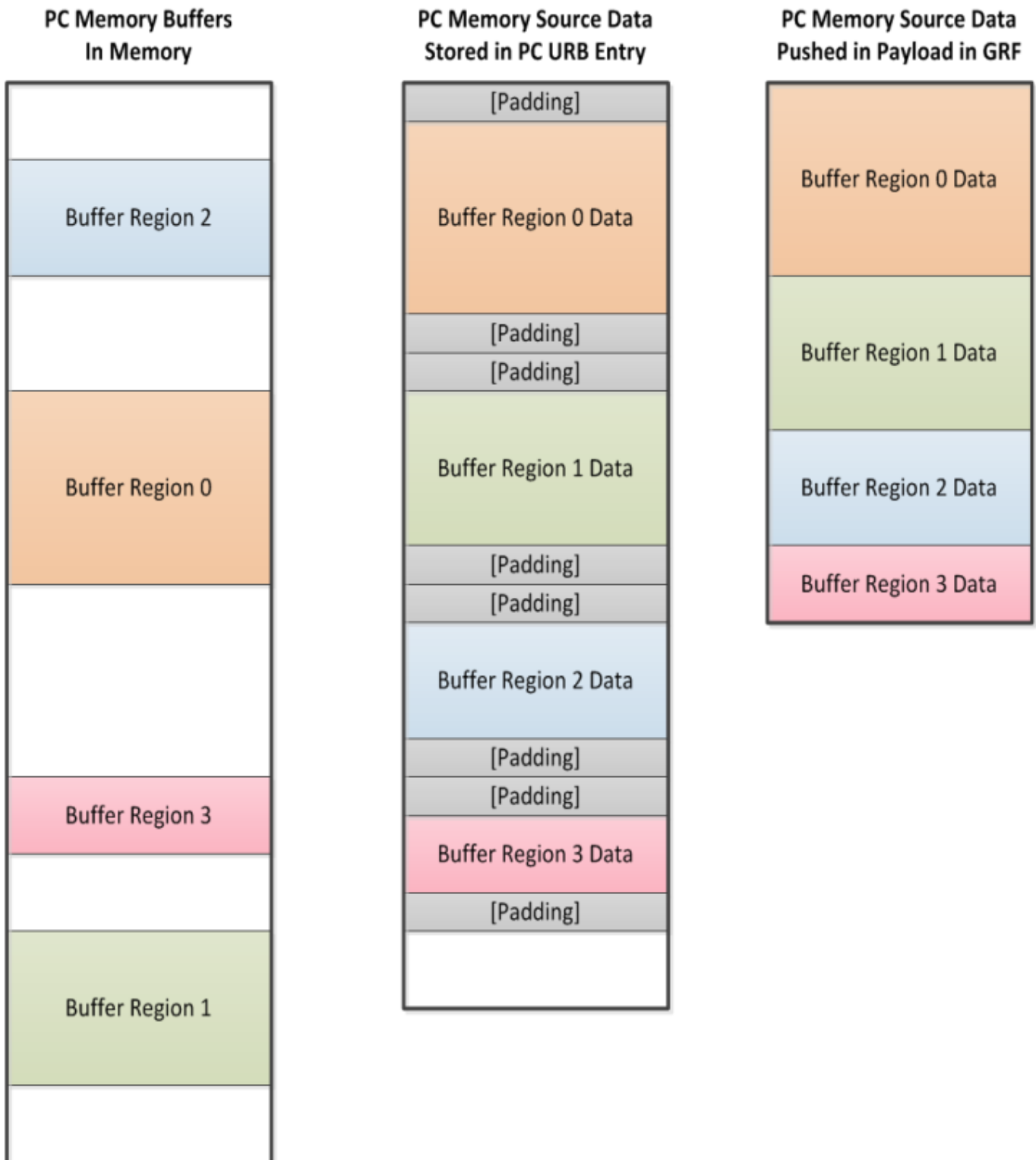
Rn		Registers prior to Push Constants
[Varies] optional	255:0	Indirect Push Constants: Push Constant data indirectly fetched from memory based on the 3DSTATE_CONSTANT_* command and read from the URB. The amount of data provided is defined by the sum of the read lengths in the last 3DSTATE_CONSTANT_*command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



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
- 32-Byte granular DynamicStateOffset from the DynamicStateBaseAddress GraphicsAddress (programmed via STATE_BASE_ADDRESS)

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 DynamicStateOffset, the **DynamicStateMemoryObjectControlState** (programmed via STATE_BASE_ADDRESS) is used and corresponding DynamicState bounds checking is performed during the memory access.

* 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 in 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
<ul style="list-style-type: none"> • 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

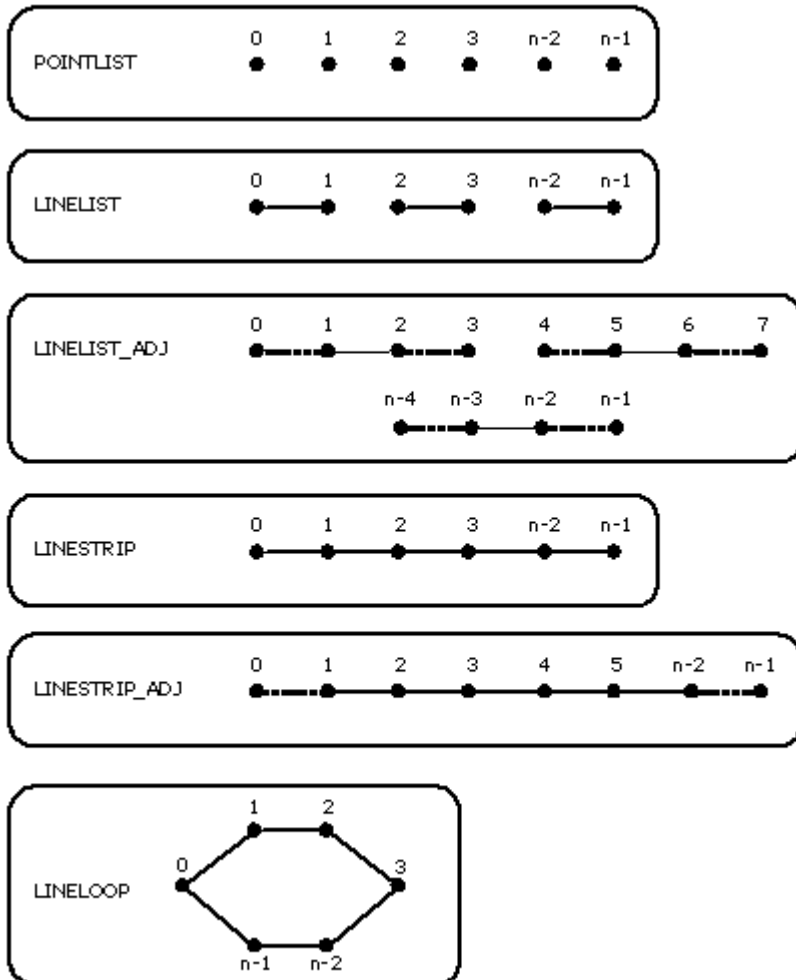
3D Primitive Topology Type (ordered alphabetically)	Description
LINELIST	<ul style="list-style-type: none"> • A list of independent line objects (2 vertices per line). • Normal usage expects a multiple of 2 vertices, though incomplete objects are silently ignored.
LINELIST_ADJ	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • A list of vertices connected such that, after the first vertex, each additional

3D Primitive Topology Type (ordered alphabetically)	Description
	<p>vertex is associated with the previous vertex to define connected line object. The first and last segments are adjacent-only vertices.</p> <ul style="list-style-type: none"> • Normal usage expects at least 4 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
LINESTRIP_BF	<ul style="list-style-type: none"> • 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_CONT	<ul style="list-style-type: none"> • 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.
POINTLIST	A list of point objects (1 vertex per point).
POINTLIST_BF	<ul style="list-style-type: none"> • 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.
POLYGON	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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. $V0=LowerRight$, $V1=LowerLeft$, $V2=UpperLeft$. Implied $V3 = V0-V1+V2$.

3D Primitive Topology Type (ordered alphabetically)	Description
	<ul style="list-style-type: none"> • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>The RECTLIST primitive is supported specifically for 2D 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:</p> <ul style="list-style-type: none"> • 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. • Viewport Mapping must be DISABLED (as is typical with the use of screen-space coordinates). </div>
RECTLIST_SUBPIXEL	The subpixel precise, axis-aligned bounding box of the object's 3 vertices is rendered.
TRIFAN	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> (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.
TRILIST	<ul style="list-style-type: none"> • A list of independent triangle objects (3 vertices per triangle). • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.

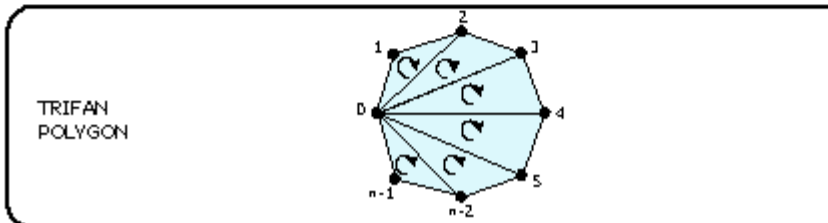
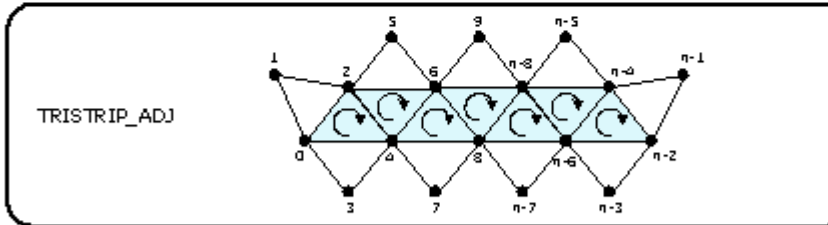
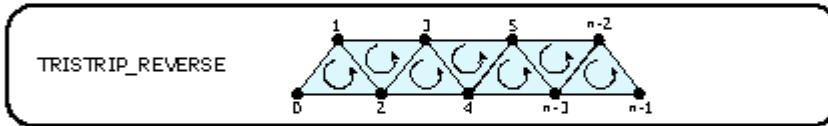
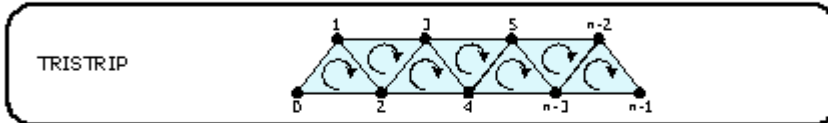
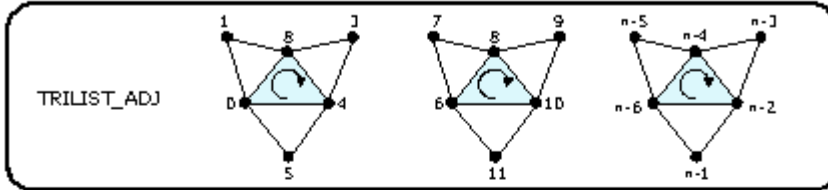
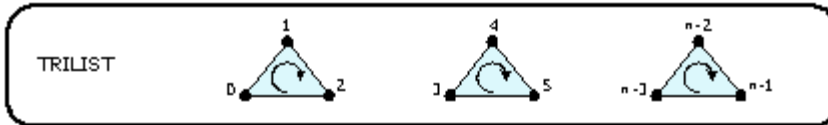
3D Primitive Topology Type (ordered alphabetically)	Description
TRILIST_ADJ	<ul style="list-style-type: none"> • A list of independent triangle objects with adjacency information (6 vertices per triangle). • Normal usage expects a multiple of 6 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread.
TRISTRIP	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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 trisrip_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	<p>Similar to TRISTRIP, though the sense of orientation (winding order) is reversed - this allows SW to break long trisrips into smaller pieces and still maintain correct face orientations.</p>
PATCHLIST_n	<p>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.)</p>

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).

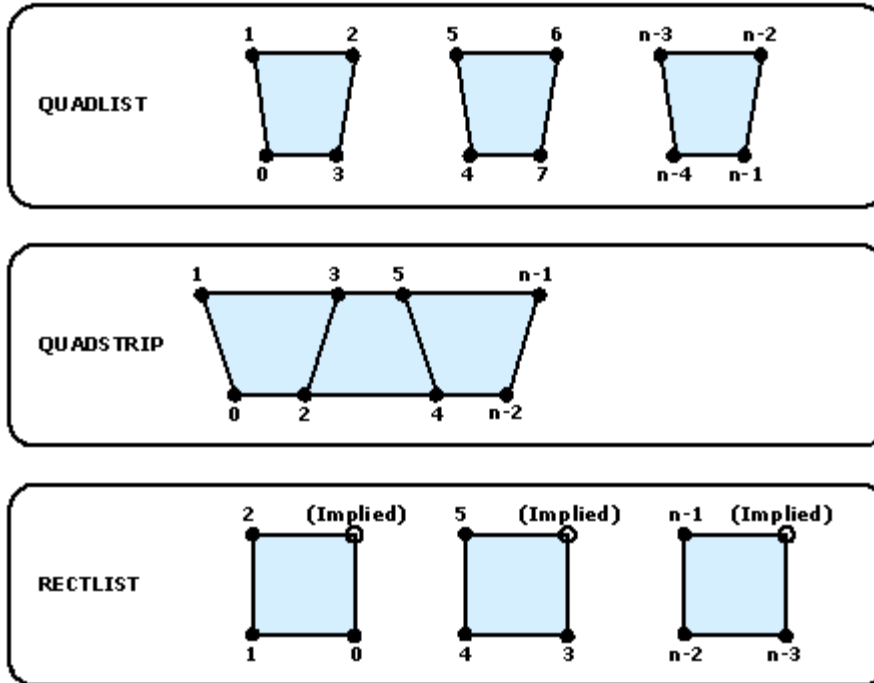


B6815-01

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).



B6816-01



B6818-01

Vertex Data Overview

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	31:0	Reserved: MBZ
D1	31:0	<p>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.</p> <p>Software is responsible for ensuring this field is zero whenever a programmable index value is not required. When a programmable index value is required</p> <p>, software must ensure that the correct 11-bit value is written to this field. Specifically, the kernels must perform a reange 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.</p> <p>Software can force an RTAIndex of 0 to be used (effectively ignoring the setting of this</p>

DWord	Bits	Description
		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
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 W 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., <u>in addition</u> 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

DWord	Bits	Description		
		<p>Vertex Position 0.XYZ 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.</p> <table border="1" data-bbox="391 457 1321 583"> <thead> <tr> <th data-bbox="391 457 1321 506">Programming Note</th> </tr> </thead> <tbody> <tr> <td data-bbox="391 506 1321 583">Replica Vertex Positions are supported in both the Render and POSH pipelines</td> </tr> </tbody> </table>	Programming Note	Replica Vertex Positions are supported in both the Render and POSH pipelines
Programming Note				
Replica Vertex Positions are supported in both the Render and POSH pipelines				
D8	31:0	<p>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.</p> <p>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.</p> <p>Format: FLOAT32</p> <p>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).</p>		
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.		
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 end 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	<p>(Remainder of Vertex Elements).</p> <p>The absolute maximum size limit on this data is specified via a maximum limit on the amount</p>		

DWord	Bits	Description
		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

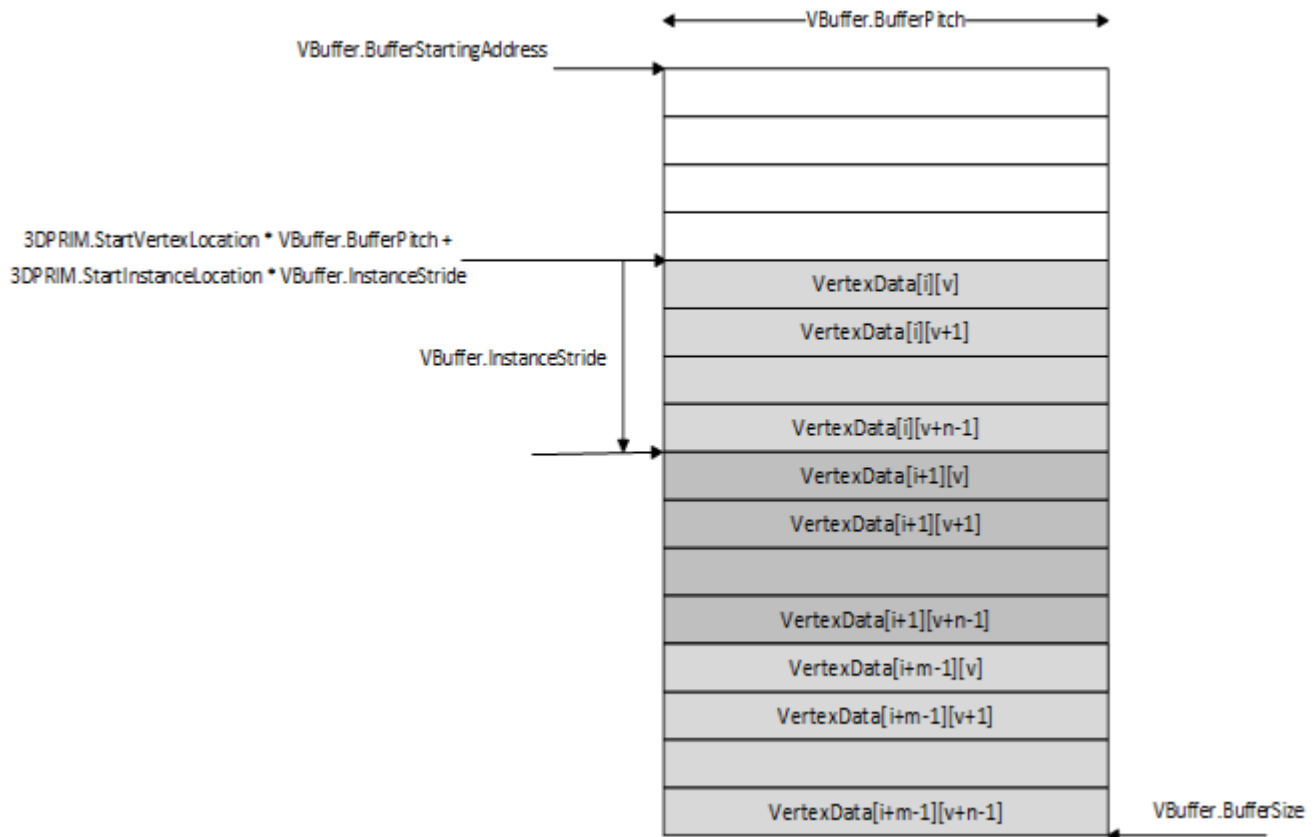
Vertex Buffers (VB) as a 2D Array

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

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



Definition	Cut Index?
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.

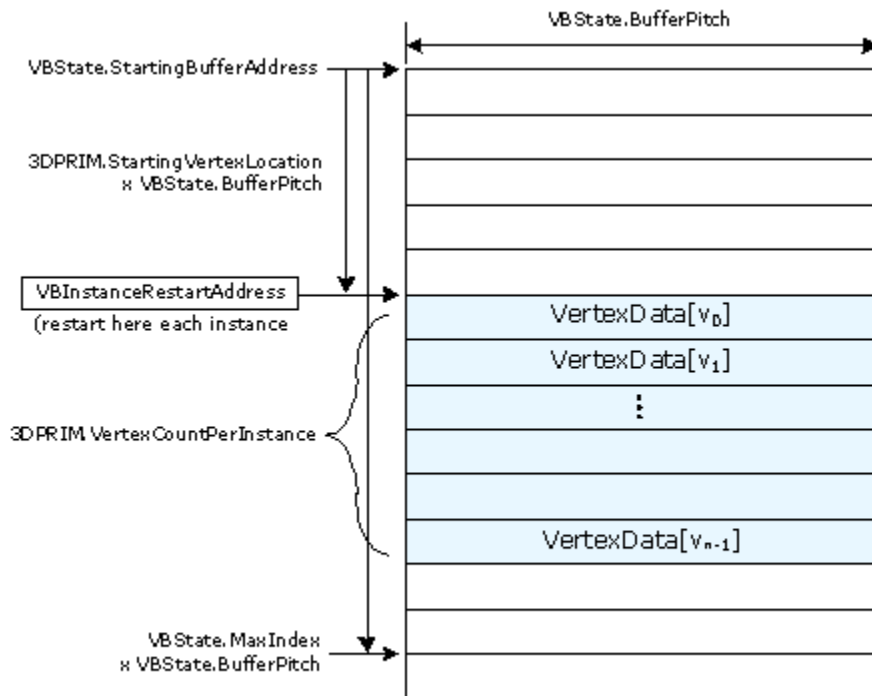
Given that the RANDOM access mode of the 3DPRIMITIVE command utilizes an IB (possibly provided by an application) to compute VB index values, VB definitions contain a **MaxIndex** value used to detect accesses beyond the end of the VBs. Any access outside the extent of a VB returns 0.

Register
3DSTATE_VERTEX_BUFFERS
VERTEX_BUFFER_STATE

VERTEXDATA Buffers - SEQUENTIAL Access

Description

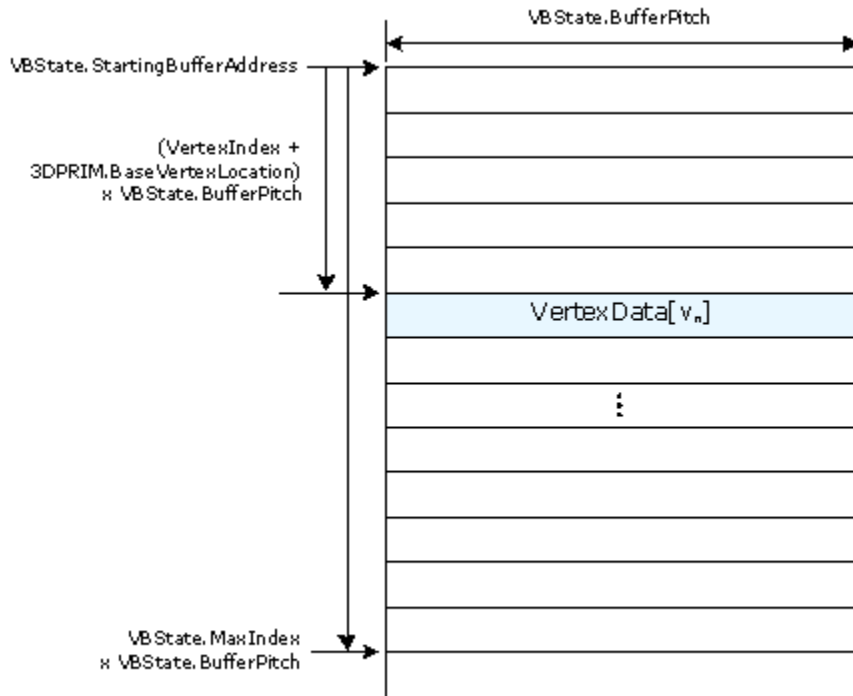
This section pertains to (a) 3DPRIMITIVE commands with VertexAccessType = SEQUENTIAL and (b) vertex elements with InstancingEnable set to DISABLED. Instead of "VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch", the address of the byte immediately beyond the last valid byte of the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



B6826-01

VERTEXDATA Buffers - RANDOM Access

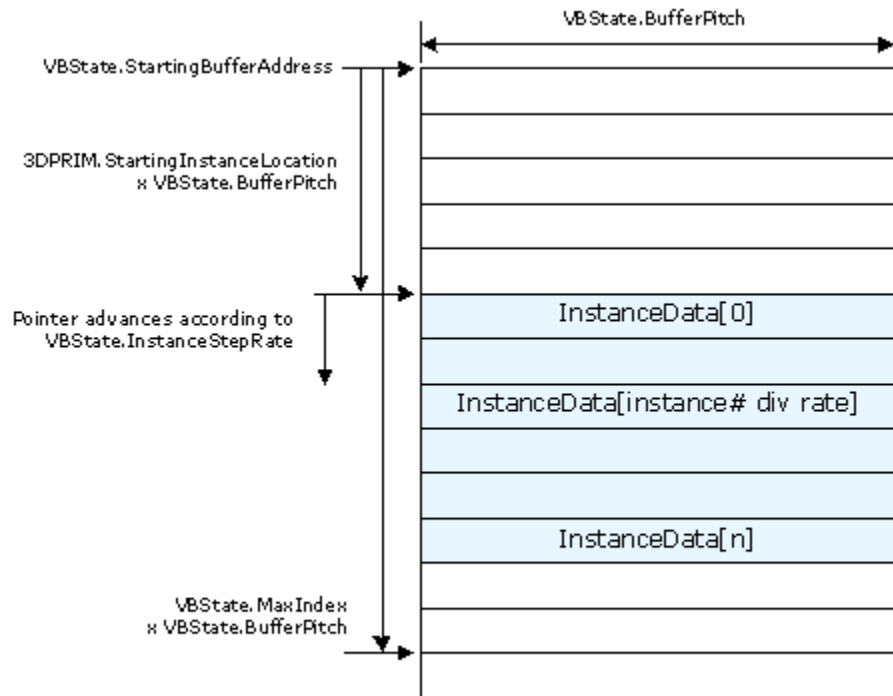
Description
<p>This section pertains to (a) 3DPRIMITIVE commands with VertexAccessType = RANDOM and (b) vertex elements with InstancingEnable set to DISABLED. Instead of "VBState.StartingBufferAddress + VBState.MaxIndex x VBState.BufferPitch", the address of the byte immediately beyond the last valid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



B6827-01

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".



B6839-01

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

Following are 3D Primitive Commands:

3DPRIMITIVE

3D Primitive Topology Type Encoding

The following table 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.

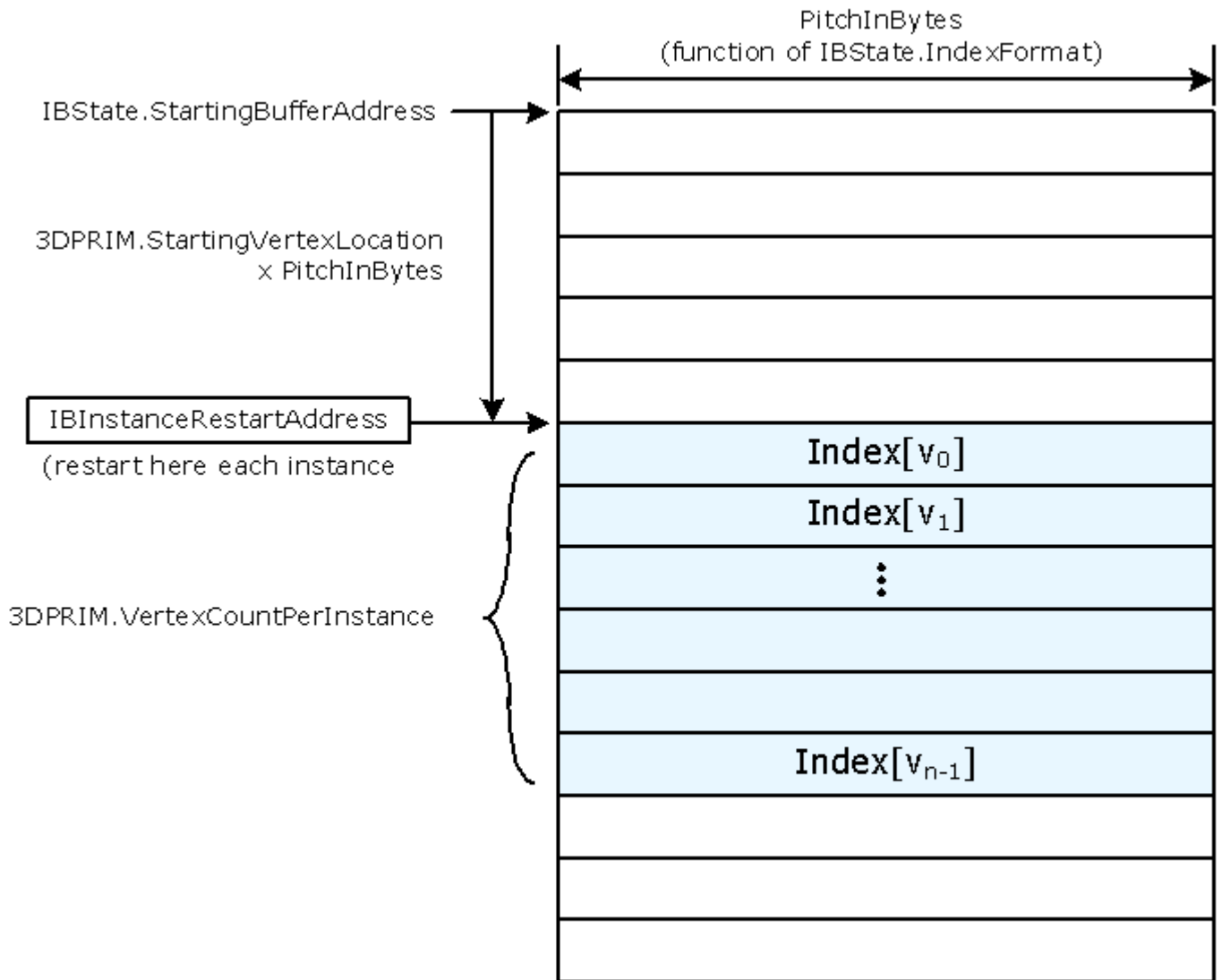
The optional insertion of VertexID and/or InstanceID into the input vertex data occurs as a separate step after the processing of VERTEX_ELEMENT structures and is controlled via the 3DSTATE_VF_SGVS command.

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

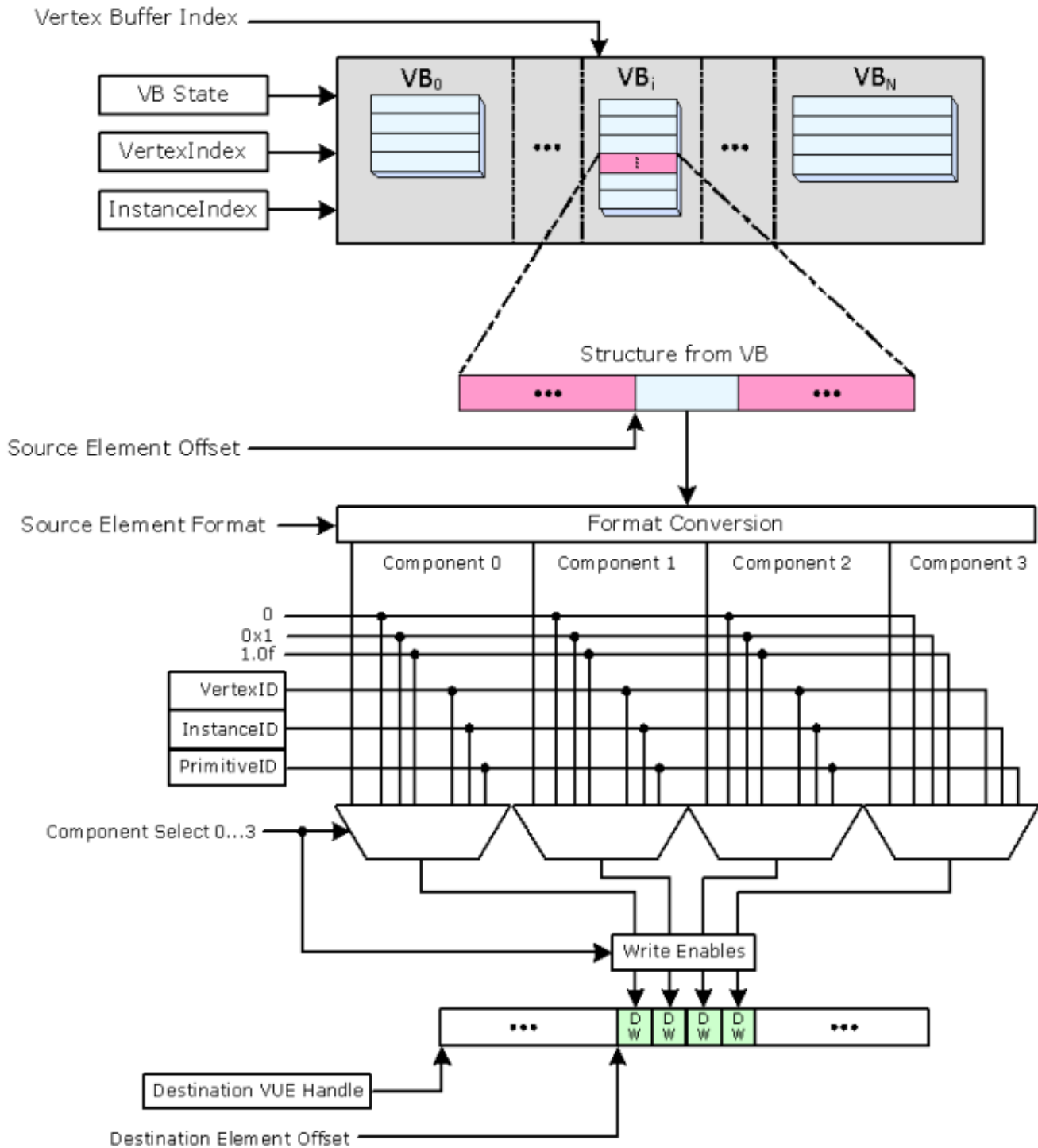
The following figure illustrates how the Index Buffer is accessed.



B6825-01

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				
	Format	0	1	2	3
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.

Note
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:

Condition
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. Usage Model: This is an exceptional condition, only required for when the VF-generated vertices contain PrimitiveID. Otherwise the VS\$ should be enabled.
	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, or (b) The input vertex contains PrimitiveID.
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

VS\$	VS Function Enable	Mode of Operation
		detected a VS shader that simply copies outputs to inputs).
	ENABLED	<p>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.</p> <p>Usage Model: Normal operation when the VS function is required and use of the VS\$ is permissible.</p>

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 1a, GPU) will be incremented for *each vertex* that is dispatched to a VS thread.

Description
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 Offset. Specifies the offset of the scratch space allocated to the thread, specified as a 1KB-granular offset from the General State Base Address . See Scratch Space Base Offset description in VS_STATE. (See <i>3D Pipeline</i> for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
R0.5	9:0	Description

DWord	Bits	Description
		<p>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.</p> <p>Reserved for HW Implementation Use.</p> <p>Format: U10</p> <p>Range: 0-727</p>
R0.4	31:5	<p>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.</p> <p>Format = SurfaceStateOffset[31:5]</p>
	4:0	Reserved
R0.3	31:5	<p>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.</p> <p>Format = DynamicStateOffset[31:5]</p>
	4	<p style="text-align: center;">Description</p> <p>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.</p>
	3:0	<p style="text-align: center;">Description</p> <p>Per Thread Scratch Space. Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two is raised to (over determine the amount of scratch space).</p> <p>(See <i>3D Pipeline</i> for further description.)</p> <p>Format = U4 power of two (in excess of 10)</p> <p>Range = [0,11] indicating [1K Bytes, 2M Bytes]</p>
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)

DWord	Bits	Description
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
[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.
		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.

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

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



DWord	Bits	Description
		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
	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]

GRF DWord	Bits	Description
	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	Per Thread Scratch Space. Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space). Programming Notes:This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it. Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31	Reserved: MBZ.
	30:24	BarrierID. This field identifies which barrier was allocated for this thread. Format: U7 Range = [0,63]
	22:16	Instance Number. A patch-relative instance number between 0 and InstanceCount-1. Format = U7
	23	Reserved
	15:0	Reserved
R0.1- R0.0	31:0	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)

GRF DWord	Bits	Description
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
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 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

GRF DWord	Bits	Description				
	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	<p>Constant Data (optional):</p> <p>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.</p>				
Varies		<p>Pushed Vertex Data (optional)</p> <p>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.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="text-align: center;">Programming Note</th> </tr> </thead> <tbody> <tr> <td style="width: 25%;">Context:</td> <td>8_PATCH Payload - Pushed Vertex Data</td> </tr> </tbody> </table> <p>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.</p>	Programming Note		Context:	8_PATCH Payload - Pushed Vertex Data
Programming Note						
Context:	8_PATCH Payload - Pushed Vertex Data					
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				

GRF DWord	Bits	Description
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...

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.

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

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 <i>even</i> 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 <i>odd</i> number of possibly unequal segments. The tessellation scheme is very similar to EVEN_FRACTIONAL partitioning, except that the edge

Partitioning Mode	Definition
	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	<p>The edge is divided into an integral number of equal segments (given some fixed-point tolerance).</p> <p>After clamping, the TF is rounded up to an integer power of 2 value. The edge is divided into that many equal segments.</p>

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:

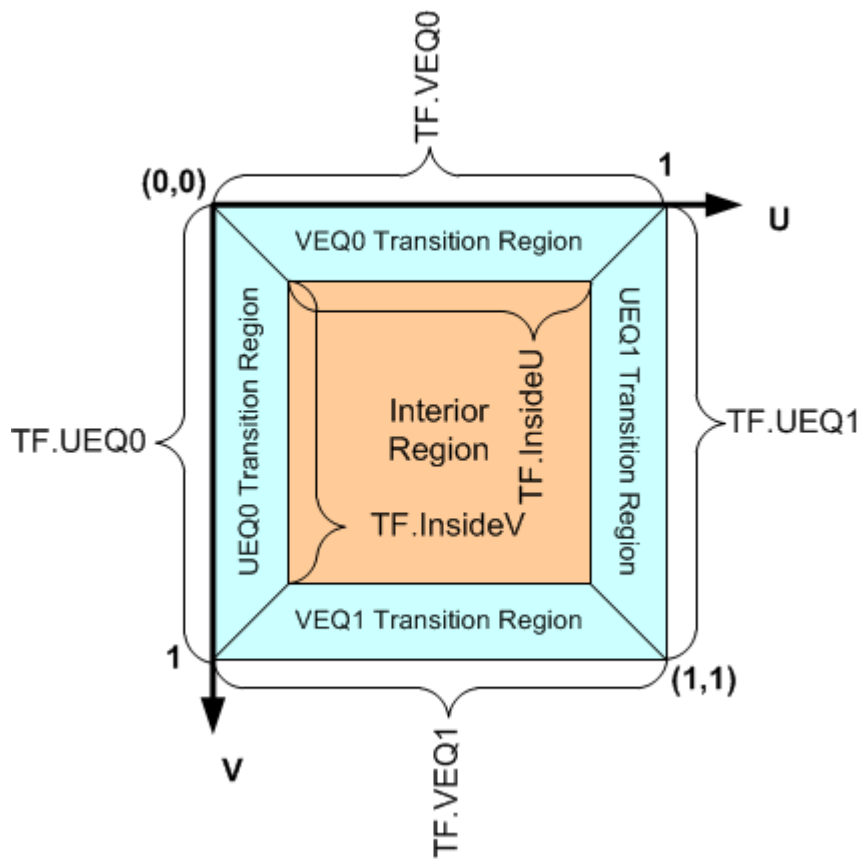
- QUAD: A square 2D region within a u,v Cartesian (rectangular) space. The region extends from the origin to u=1 and v=1. Within the region, tessellation domain locations are determined. The possible output topologies include points, clockwise triangles, and counterclockwise triangles.
- TRI: A triangular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

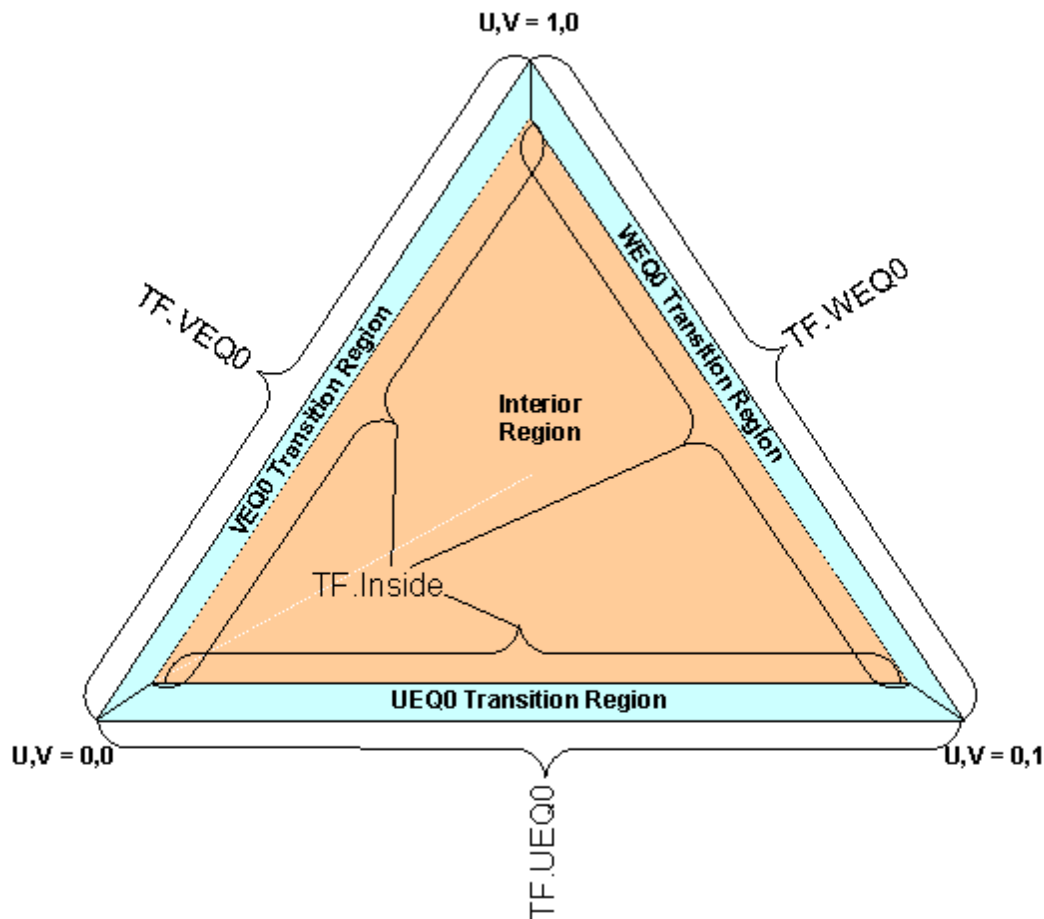
QUAD Domain



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.

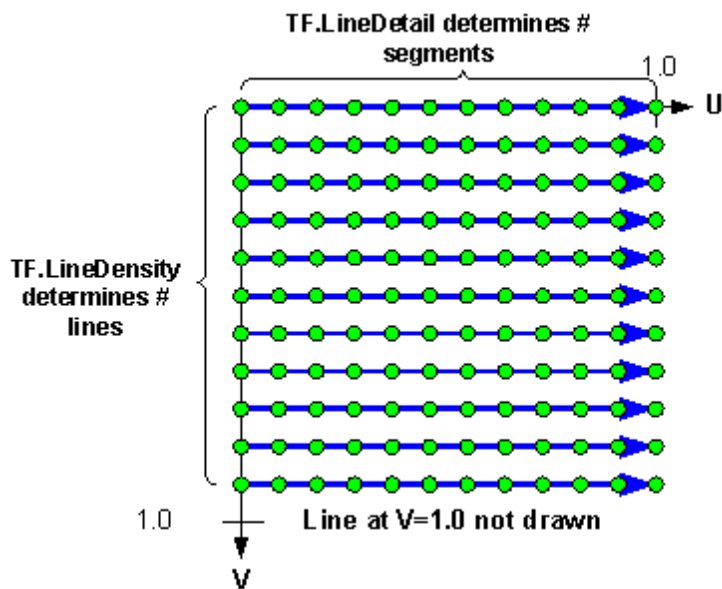
TRI Domain



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



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.

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

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	<p>Thread ID. This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time.</p> <p>Format: Reserved for HW Implementation Use.</p>
R0.5	31:10	<p>Scratch Space Offset. Specifies the offset of the scratch space allocated to the thread, specified as a 1KB-granular offset from the General State Base Address. See Scratch Space Base Offset description in VS_STATE.</p> <p>(See <i>3D Pipeline</i> for further description on scratch space allocation).</p> <p>Format = GeneralStateOffset[31:10]</p>
	9:0	<p>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.</p> <p>Format: Reserved for HW Implementation Use.</p>
R0.4	31:5	<p>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.</p> <p>Format = SurfaceStateOffset[31:5]</p>
	4:0	Reserved
R0.3	31:5	<p>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.</p> <p>Format = DynamicStateOffset[31:5]</p>
	4	Reserved
	3:0	<p>Per Thread Scratch Space. Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space).</p> <p>Format = U4 power of two (in excess of 10)</p> <p>Range = [0,11] indicating [1K Bytes, 2M Bytes]</p>

DWord	Bits	Description				
R0.2	31:0	Reserved: delivered as zeros (reserved for message header fields)				
R0.1	31:0	<p>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.</p> <p>Format: U32</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="background-color: #e6f2ff;">Programming Note</th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;">The contents of this field is UNDEFINED if the 3DSTATE_DS::PrimitiveIDNotRequired state bit is set.</td> <td></td> </tr> </tbody> </table>	Programming Note		The contents of this field is UNDEFINED if the 3DSTATE_DS::PrimitiveIDNotRequired state bit is set.	
Programming Note						
The contents of this field is UNDEFINED if the 3DSTATE_DS::PrimitiveIDNotRequired state bit is set.						
R0.0	31:27	Reserved				
	26:16	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="background-color: #e6f2ff;">Description</th> </tr> </thead> <tbody> <tr> <td>Reserved</td> </tr> </tbody> </table>	Description	Reserved		
	Description					
Reserved						
15:0	<p>Patch URB Offset. This is the offset within the URB where the patch data is stored.</p> <p>Format: U14 64B-granular offset into the URB</p>					
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	<p>Domain Point 0 U Coordinate. U coordinate associated with Domain Point 0.</p> <p>Format: FLOAT32</p>				
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	<p>Domain Point 0 V Coordinate. V coordinate associated with Domain Point 0.</p> <p>Format: FLOAT32</p>				
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.)				

DWord	Bits	Description
R3.4	31:0	Domain Point 4 W Coordinate. (See Domain Point 0 W Coordinate.)
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
	15:0	Domain Point 0 URB Return Handle. This is the offset within the URB where domain point 0 is to be stored. Format: U14 64B-granular offset into the 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
R0.6	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	<p>Scratch Space Offset. Specifies the offset of the scratch space allocated to the thread, specified as a 1KB-granular offset from the General State Base Address. See Scratch Space Base Offset description in VS_STATE.</p> <p>(See <i>3D Pipeline</i> for further description on scratch space allocation).</p> <p>Format = GeneralStateOffset[31:10]</p>		
	9:0	<p>FTTID. 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.</p> <p>Format: Reserved for HW Implementation Use.</p>		
R0.4	31:5	<p>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.</p> <p>Format = SurfaceStateOffset[31:5]</p>		
	4:0	Reserved		
R0.3	31:5	<p>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.</p> <p>Format = DynamicStateOffset[31:5]</p>		
	4	Reserved		
	3:0	<p>Per Thread Scratch Space. Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space).</p> <p>Format = U4 power of two (in excess of 10)</p> <p>Range = [0,11] indicating [1K Bytes, 2M Bytes]</p>		
R0.2	31:0	Reserved: delivered as zeros (reserved for message header fields)		
R0.1	31:27	Reserved		
	26:16	<table border="1"> <tr> <td>Description</td> </tr> <tr> <td>Reserved</td> </tr> </table>	Description	Reserved
	Description			
Reserved				
15:0	<p>Patch 1 URB Offset. This is the offset within the URB where the Patch 1 data is stored.</p> <p>Format: U16 64B-granular offset into the URB</p>			
R0.0	31:27	Reserved		
	26:16	<table border="1"> <tr> <td>Description</td> </tr> <tr> <td>Reserved</td> </tr> </table>	Description	Reserved
Description				
Reserved				

DWord	Bits	Description
	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
		The following "R1" GRF is not included in the payload if 3DSTATE_DS::PrimitiveIDNotRequired is set.
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.)

DWord	Bits	Description
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 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
[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.

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 section 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

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] = (<u>0</u>); [1] = (<u>1</u>); ...; [N-2] = (<u>N-2</u>);	
POINTLIST_BF	N/A	
LINELIST (N is multiple of 2)	[0] = (0, <u>1</u>); [1] = (2, <u>3</u>); ...; [(N/2)-1] = (N-2, <u>N-1</u>)	
LINELIST_ADJ (N is multiple of 4)	[0] = (0,1, <u>2</u> ,3); [1] = (4,5, <u>6</u> ,7); ...; [(N/4)-1] = (N-4,N-3, <u>N-2</u> ,N-1)	
LINESTRIP (N >= 2)	[0] = (0, <u>1</u>); [1] = (1, <u>2</u>); ...; [N-2] = (N-2, <u>N-1</u>)	
LINESTRIP_ADJ, LINESTRIP_ADJ_CONT (N >= 4)	[0] = (0,1, <u>2</u> ,3); [1] = (1,2, <u>3</u> ,4); ...; [N-4] = (N-4,N-3, <u>N-2</u> ,N-1)	LINESTRIP_ADJ_CONT is added 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	

PrimTopologyType	Order of Vertices in Payload	
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...); [{modified PrimType passed to thread}]	GS Notes
LINELOOP (N >= 2)	[0] = (0, <u>1</u>); [1] = (1, <u>2</u>); [N] = (N-1, <u>0</u>);	Not supported after GS.
TRILIST (N is multiple of 3)	[0] = (0,1, <u>2</u>); [1] = (3,4, <u>5</u>); ...; [(N/3)-1] = (N-3,N-2, <u>N-1</u>)	
RECTLIST, RECTLIST_SUBPIXEL	Same as TRILIST	Handled same as TRILIST
TRILIST_ADJ (N is multiple of 6)	[0] = (0,1,2,3, <u>4</u> ,5); [1] = (6,7,8,9, <u>10</u> ,11); ...; [(N/6)-1] = (N-6,N-5,N-4,N-3, <u>N-2</u> ,N-1)	
TRISTRIP (Reorder Leading) (N >= 3)	[0] = (0,1, <u>2</u>); {TRISTRIP} [1] = (1, <u>3</u> ,2); {TRISTRIP_REV} [k even] = (k,k+1, <u>k+2</u>) {TRISTRIP} [k odd] = (k, <u>k+2</u> ,k+1) {TRISTRIP_REV} [N-3] = (see above)	"Odd" triangles have vertices reordered and identified as TRISTRIP to inform the thread.
TRISTRIP (Reorder Trailing) (N >= 3)	[0] = (0,1, <u>2</u>) {TRISTRIP} [1] = (2,1, <u>3</u>) {TRISTRIP_REV}; ... [k even] = (k,k+1, <u>k+2</u>) {TRISTRIP} [k odd] = (k+1,k, <u>k+2</u>) {TRISTRIP_REV} [N-3] = (see above)	"Odd" triangles have vertices reordered and identified as TRISTRIP_REV to inform the thread.
TRISTRIP_REV (Reorder	[0] = (0, <u>2</u> ,1)	"Even" triangles have vertices reordered and identified as

PrimTopologyType	Order of Vertices in Payload	GS Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...); [{modified PrimType passed to thread}]	
<u>Leading)</u> (N >= 3)	{TRISTRIP_REV}; [1] = (1,2, <u>3</u>) {TRISTRIP}; ...; [k even] = (k, <u>k+2</u> ,k+1) {TRISTRIP_REV} [k odd] = (k,k+1, <u>k+2</u>) {TRISTRIP} [N-3] = (see above)	TRISTRIP to inform the thread.
TRISTRIP_REV (<u>Reorder Trailing</u>) (N >= 3)	[0] = (1,0, <u>2</u>) {TRISTRIP_REV} [1] = (1,2, <u>3</u>) {TRISTRIP}; ...; [k even] = (k+1,k, <u>k+2</u> ,) {TRISTRIP_REV} [k odd] = (k,k+1, <u>k+2</u>) {TRISTRIP} [N-3] = (see above)	"Even" triangles have vertices reordered and identified as TRISTRIP_REV to inform the thread.
TRISTRIP_ADJ (<u>Reorder Leading</u>) (N >= 6)	N = 6 or 7: [0] = (0,1,2,5, <u>4</u> ,3) N = 8 or 9: [0] = (0,1,2,6, <u>4</u> ,3); [1] = (2,5, <u>6</u> ,7,4,0); ...; N >= 10: [0] = (0,1,2,6, <u>4</u> ,3); [1] = (2,5, <u>6</u> ,8,4,0); ...; [k>1, even] = (2k,2k-2, 2k+2, 2k+6, <u>2k+4</u> , 2k+3); [k>2, odd] = (2k, 2k+3, <u>2k+4</u> , 2k+6, 2k+2, 2k-2);...; Trailing object: [(N/2)-3, even] = (N-6,N-8,N-4,N-1, <u>N-2</u> ,N-3);	Objects have vertices reordered.

PrimTopologyType	Order of Vertices in Payload	
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...); [{modified PrimType passed to thread}]	GS Notes
	[(N/2)-3, odd] = (N-6, N-3, <u>N-2</u> , N-1, N-4, N-8);	
TRISTRIP_ADJ (<u>Reorder Trailing</u>) (N >= 6)	N = 6 or 7: [0] = (0, 1, 2, 5, <u>4</u> , 3) N = 8 or 9: [0] = (0, 1, 2, 6, <u>4</u> , 3); [1] = (4, 0, 2, 5, <u>6</u> , 7); ...; N >= 10: [0] = (0, 1, 2, 6, <u>4</u> , 3); [1] = (4, 0, 2, 5, <u>6</u> , 8); ...; [k > 1, even] = (2k, 2k-2, 2k+2, 2k+6, <u>2k+4</u> , 2k+3); [k > 2, odd] = (2k+2, 2k-2, 2k, 2k+3, <u>2k+4</u> , 2k+6); ...; Trailing object: [(N/2)-3, even] = (N-6, N-8, N-4, N-1, <u>N-2</u> , N-3); [(N/2)-3, odd] = (N-4, N-8, N-6, N-3, <u>N-2</u> , 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).
TRIFAN (N > 2)	[0] = (0, 1, <u>2</u>); [1] = (0, 2, <u>3</u>); ...; [N-3] = (0, N-2, <u>N-1</u>);	
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON, POLYGON_CONT	Same as TRIFAN	POLYGON_CONT is added POLYGON_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
QUADLIST	[0] = (0, 1, 2, <u>3</u>); [1] = (4, 5, 6, <u>7</u>); ...; [(N/4)-1] = (N-4, N-3, N-2, <u>N-1</u>);	Not supported after GS. QUADLIST primitives are converted into POLYGONS in VF, and therefore never reach the GS.

PrimTopologyType	Order of Vertices in Payload	GS Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...); [{modified PrimType passed to thread}]	
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	[0] = (0);
PATCHLIST_2	[1] = (1); ...;
PATCHLIST_3..32	[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):

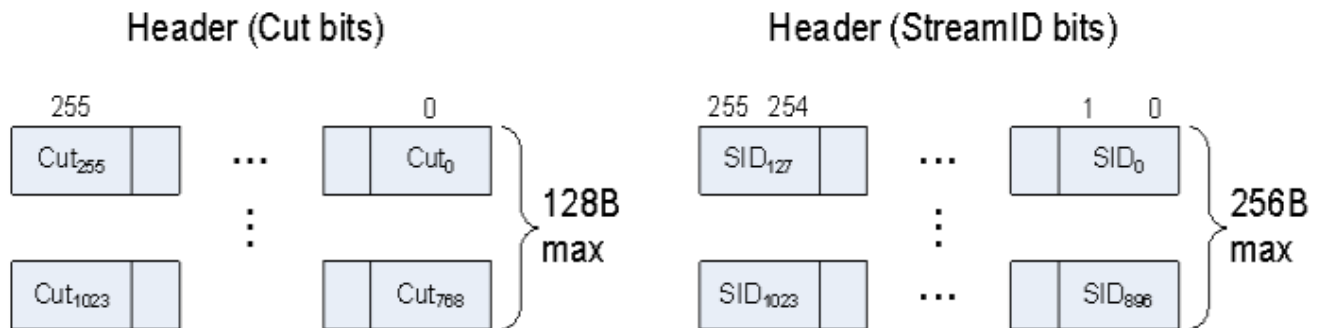
- Compiled application-provided "GS shader" programs, specifying an algorithm to convert the vertices of an input object into some output primitives. For example, a GS shader may convert lines of a line strip into polygons representing a corresponding segment of a blade of grass centered on the line. Or it could use adjacency information to detect silhouette edges of triangles and output polygons extruding out from 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, LINSTRIIP, 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 $\leq 16B$ 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 $4 \times 16B = 64B/\text{vertex}$. You have to have room for $1024 \text{ scalars} / 3 \text{ scalar/vtx} = 341 \text{ vertices}$. $341 \times 64B = 21,824B$. Then add 96B to hold $2b/\text{vtx}$ 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 $64B/\text{vertex}$ (16B vtx header, 16B position, 16B for the 2nd element, and 16B of pad). You have to have room for $1024 \text{ scalars} / 5 = 204 \text{ vertices}$. $204 \times 64 = 13,056B$. Then add 64B to hold $2b/\text{vtx}$ 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

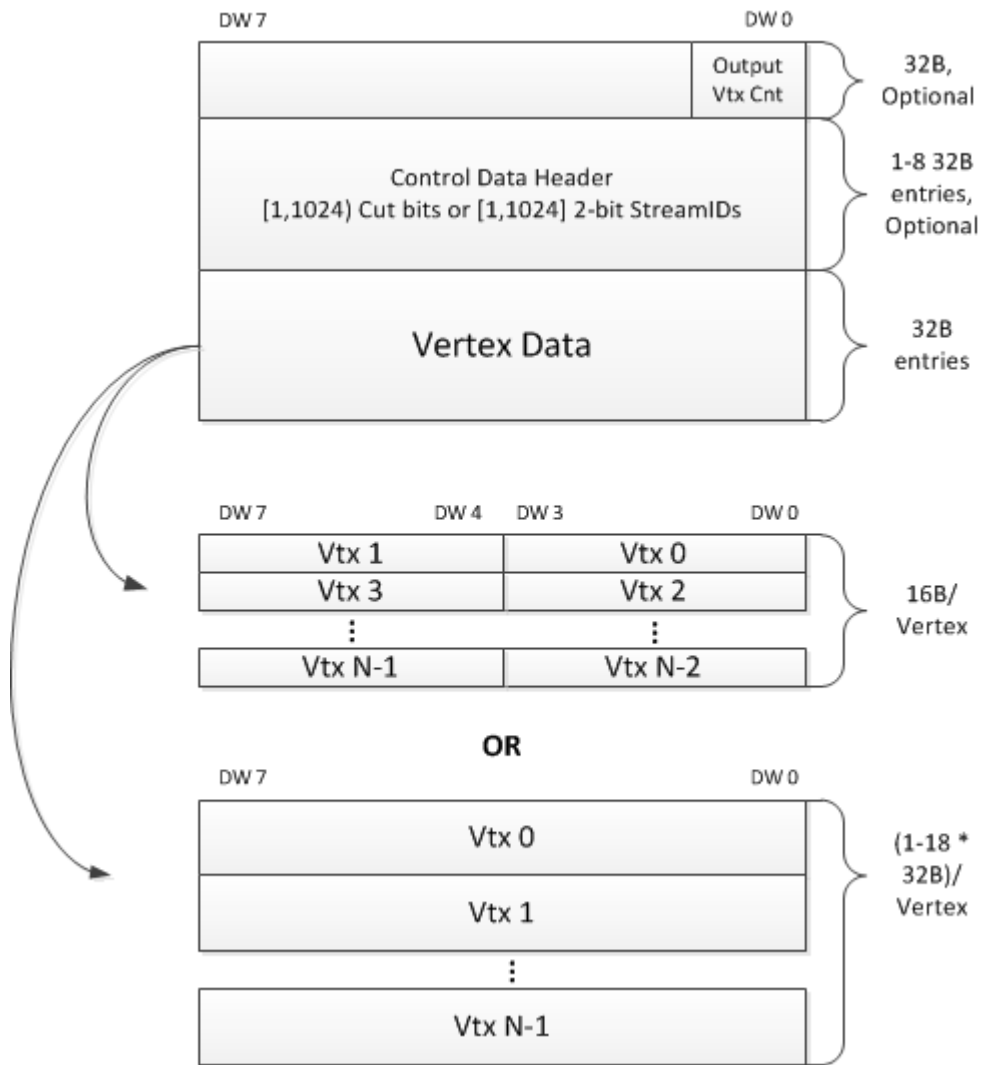
The GS URB entry is the same as in the two previous generations with the following exception: If **Static Output** (3DSTATE_GS) is clear, the URB entry starts with a 32B OUTPUT_VERTEX_COUNT structure as defined below. The control header (if present) immediately follows this structure. If **Static Output**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

PrimTopologyType	Supported for GS Thread Output?
POINTLIST	Yes
POINTLIST_BF	No
POLYGON	Yes
QUADLIST	No
QUADSTRIP	No
RECTLIST	Yes
TRIFAN	Yes
TRIFAN_NOSTIPPLE	Yes
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.

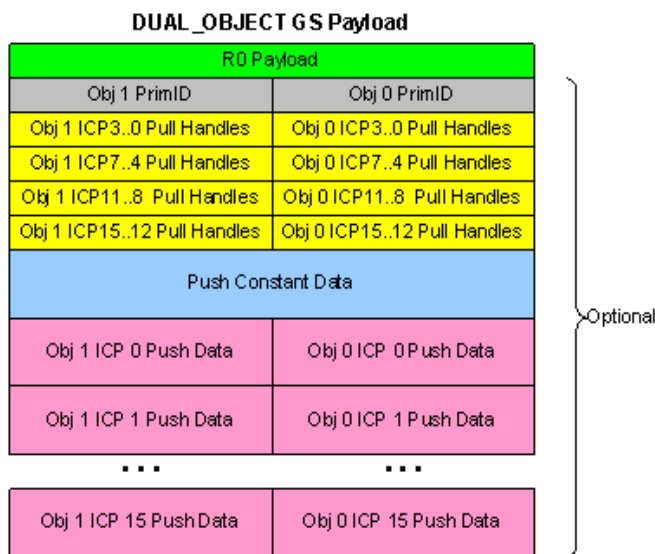
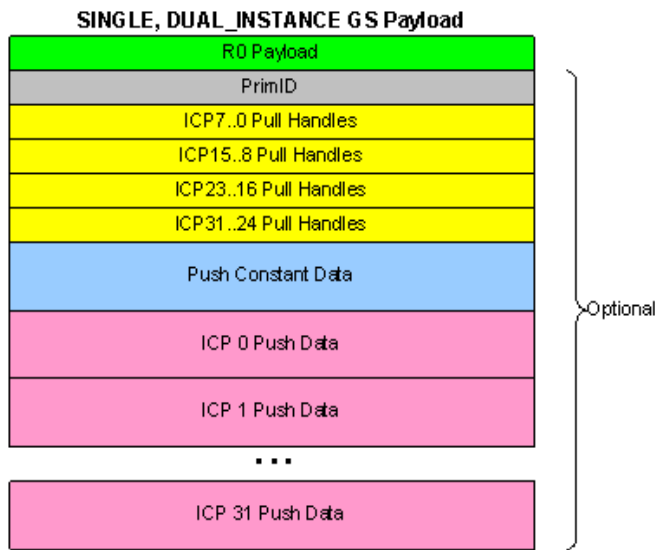
Refer to the *Statistics Gathering* summary provided earlier in this specification. Refer to the *Memory Interface Registers* chapter for details on these MMIO pipeline statistics counter registers, as well as the chapters corresponding to the other functions involved (e.g., DataPort, URB shared functions).

Payloads

Thread Payload High-Level Layout

Thread Payload High-Level Layout 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 enable 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 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]
	9:0	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	Per Thread Scratch Space. Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two is raised to (over determine the amount of scratch space). (See Generic Pipeline Stage for further description). Programming Notes: This amount is available to the kernel for information only. It is passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port ignores it.

GRF DWord	Bits	Description
		Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
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. 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-R0.0	31:0	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
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)

GRF DWord	Bits	Description
		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) 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

GRF DWord	Bits	Description
Rn.1	31:16	Reserved.
	15:0	Object 1 ICP 0 Handle
Rn.0	31:16	Reserved.
	15:0	Object 0 ICP 0 Handle
[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,

GRF DWord	Bits	Description
		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.
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.

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	<p>If clear, the SO stage will not forward any topologies down the pipeline.</p> <p>If set, the SO stage will forward topologies associated with Render Stream Select down the pipeline.</p> <p>This bit is used even if SO Function Enable is DISABLED.</p>	<pre>= (3DSTATE_STREAMOUT::Force_Rendering == Force_On) ((3DSTATE_STREAMOUT::Force_Rendering != 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 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))</pre>

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.			

DW1[21]	DW1[20]	Stream Offset	Action
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	Order of Vertices Streamed Out	Any SOL Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>, ...);	
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)	

PrimTopologyType	Order of Vertices Streamed Out	Any SOL Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...);	
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 added 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-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	[0] = (1,0,2)	"Even" triangles have vertices reordered to yield increasing

PrimTopologyType	Order of Vertices Streamed Out	Any SQL Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...);	
(N >= 3) REORDER_TRAILING	[1] = (1,2,3);...; [k even] = (k+1,k,k+2) [k odd] = (k,k+1,k+2) [N-3] = (see above)	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: [(N/2)-3, even] = (N- 6,N-4,N-2); [(N/2)-3, odd] = (N- 6,N-2,N-4);	"Odd" objects have vertices reordered to yield increasing-by-2 leading vertices starting with v0.
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);	

PrimTopologyType	Order of Vertices Streamed Out	Any SOL Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,...);	
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON, POLYGON_CONT	Same as TRIFAN	POLYGON_CONT is added. 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 $(\text{write_offset} + \text{vertex_count} * \text{buffer_pitch})$. 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
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
<code>3DSTATE_VIEWPORT_STATE_POINTERS_CC</code>
<code>3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP</code>
<code>3DSTATE_SCISSOR_STATE_POINTERS</code>
<code>3DSTATE_RASTER</code>

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, VPIIndex, 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 counterclockwise (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	<p>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).</p> <p>DX10 allows simultaneous use of ClipDistance and Cull Distance test of up to 8 distances.</p> <p>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.</p>	<pre> = (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 ? 3DSTATE_VS:: VS User Clip Distance Cull Test Enable Bitmask : 0 </pre>
CLIP_INT::User Clip Distance Clip Test Enable Bitmask	<p>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).</p> <p>DX10 allows simultaneous use of ClipDistance and Clip Distance test of up to 8 distances.</p> <p>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.</p>	<pre> = (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 : </pre>

Signal	Description	Formula
		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_YMIN)
        outCode[VP_YMAX] = (y <= vp_YMAX)
    }
#endif
}
}
```



```

}
#endif
if (rhw_neg) {
    outCode[VP_XMIN] = (x > vp_XMIN)
    outCode[VP_XMAX] = (x < vp_XMAX)
    outCode[VP_YMIN] = (y > vp_YMIN)
    outCode[VP_YMAX] = (y < vp_YMAX)
}
}

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(z)) {
    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. This 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 and 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 added LINESTRIP_ADJ_CONT is generated by the Vertex Fetch unit

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
		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 added POLYGON_CONT is generated by the Vertex Fetch unit on a restore of a mid-draw pre-empted 3DPRIMITIVE.
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., RECLISTS) 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 <i>3D Pipeline</i> 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).

Variable	Type	Description
		<p>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.</p> <p>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.</p>
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	<p>Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded.</p> <p>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.</p>
viewPortIndex	uint	<p>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.</p> <p>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 within the topology.</p>
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
- VPIIndex, 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. *Primitive Assembly*. 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	≥ 2

primType	VertexCount Restriction
3DPRIM_LINESTRIP_CONT_BF	
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
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 <u>object</u> 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. <i>Primitive Assembly</i>
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.

Variable	Type	Description
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

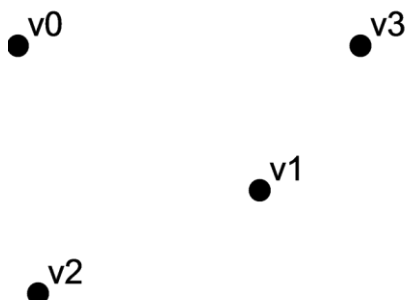
PrimTopologyType	Viewport Index Usage
POINTLIST POINTLIST_BF	Each vertex supplies the VPIndex for the corresponding point object
LINELIST	The leading vertex of each line supplies the VPIndex for the corresponding line object. V0.VPIndex-> Line(V0,V1) V2.VPIndex-> Line(V2,V3) ...
LINESTRIP LINESTRIP_BF 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.

PrimTopologyType	Viewport Index Usage
TRILIST RECTLIST	<p>The leading vertex of each triangle/rect supplies the VPIndex for the corresponding triangle/rect objects.</p> <p>V0.VPIndex-> Tri(V0,V1,V2)</p> <p>V3.VPIndex-> Tri(V3,V4,V5)</p> <p>...</p> <p>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 outputting multiple viewport indices for other topologies either autostrip needs to be disable or clipper guardband test needs to be disabled.</p>
TRISTRIP TRISTRIP_REVERSE	<p>The leading vertex of each triangle supplies the VPIndex for the corresponding triangle object.</p> <p>V0.VPIndex-> Tri(V0,V1,V2)</p> <p>V1.VPIndex-> Tri(V1,V2,V3)</p> <p>...</p> <p>NOTE: If the VPIndex changes within the primitive, shared vertices will be processed (mapped) multiple times.</p>
TRIFAN TRIFAN_NOSTIPPLE POLYGON	<p>The first vertex (V0) supplies the VPIndex for all triangle objects.</p>

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
  }

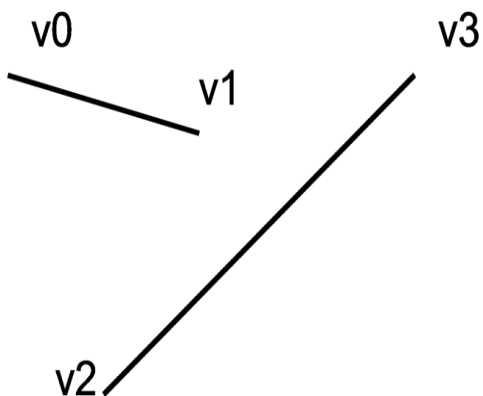
  for each (vertex in [0..vertexCount-1])
  {
    v[0] <- vIn[i] // copy all arrays
                  // (for example, v[]X, v[]Y, and so on)
    ObjectSetup()
  }
}

```

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.

```

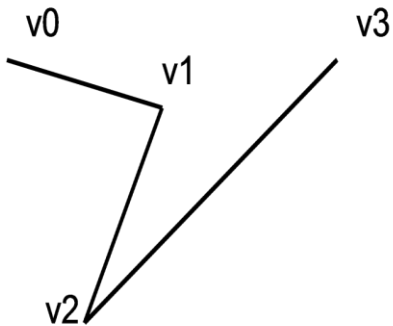
LineListDecomposition()
{
  objectType = 3DOBJ_LINE
  nV = 2
  provokingVtx = Line List/Strip Provoking Vertex Select      continueStipple = FALSE
  for each (vertex I in [0..vertexCount-2] by 2)
  {
    v[0] arrays <- vIn[i] arrays
    v[1] arrays <- vIn[i+1] arrays
    ObjectSetup()
  }
}

```

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.

```

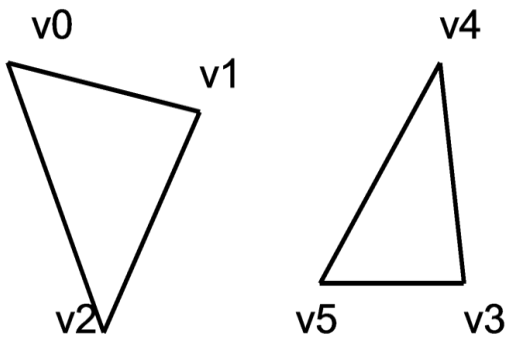
LineStripDecomposition()
{
  objectType = 3DOBJ_LINE
  nV = 2
  provokingVtx = Line List/Strip Provoking Vertex Select
  if (primType == 3DPRIM_LINESTRIP)
  {
    backFacing = FALSE
    continueStipple = FALSE
  } else if (primType == 3DPRIM_LINESTRIP_BF)
  {
    backFacing = TRUE
    continueStipple = 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



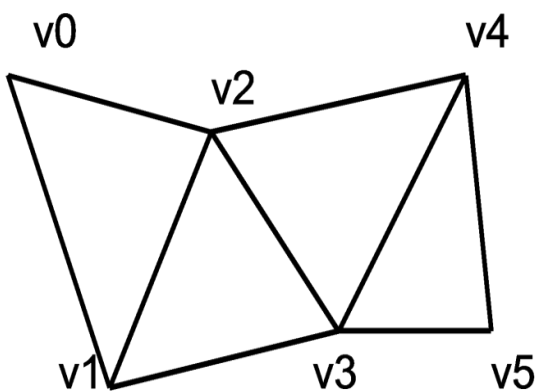
The decomposition process divides the list into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v3,v4,v5; and so on. The *provokingVertex* of each object is 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 (counterclockwise), 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.

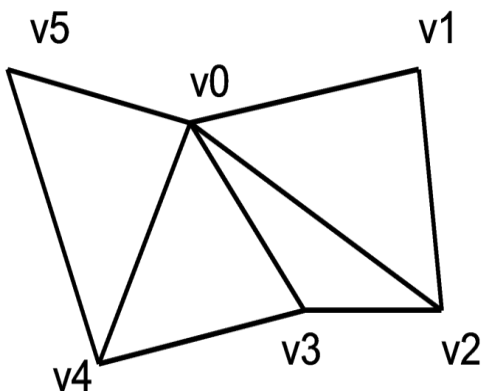
To accommodate the situation where the driver is forced to break an input strip primitive into multiple tristrip primitive commands (for example, due to ring or batch buffer size restrictions), two tristrip primitive types are supported. 3DPRIM_TRISTRIP is used for the initial section of a strip, and 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()
    }
}
```

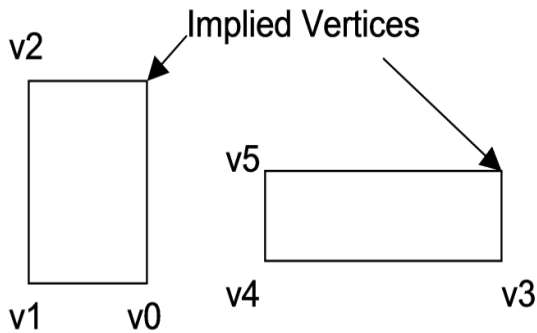
Polygon Decomposition

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)

Invalid Position Culling (Pre/Post-Transform)

At input 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:

$$x' = \mathbf{m00} * x + \mathbf{m30}$$

$$y' = \mathbf{m11} * y + \mathbf{m31}$$

$$z' = \mathbf{m22} * z + \mathbf{m32}$$

Software is responsible for computing the matrix elements from the viewport information provided to it from the API.

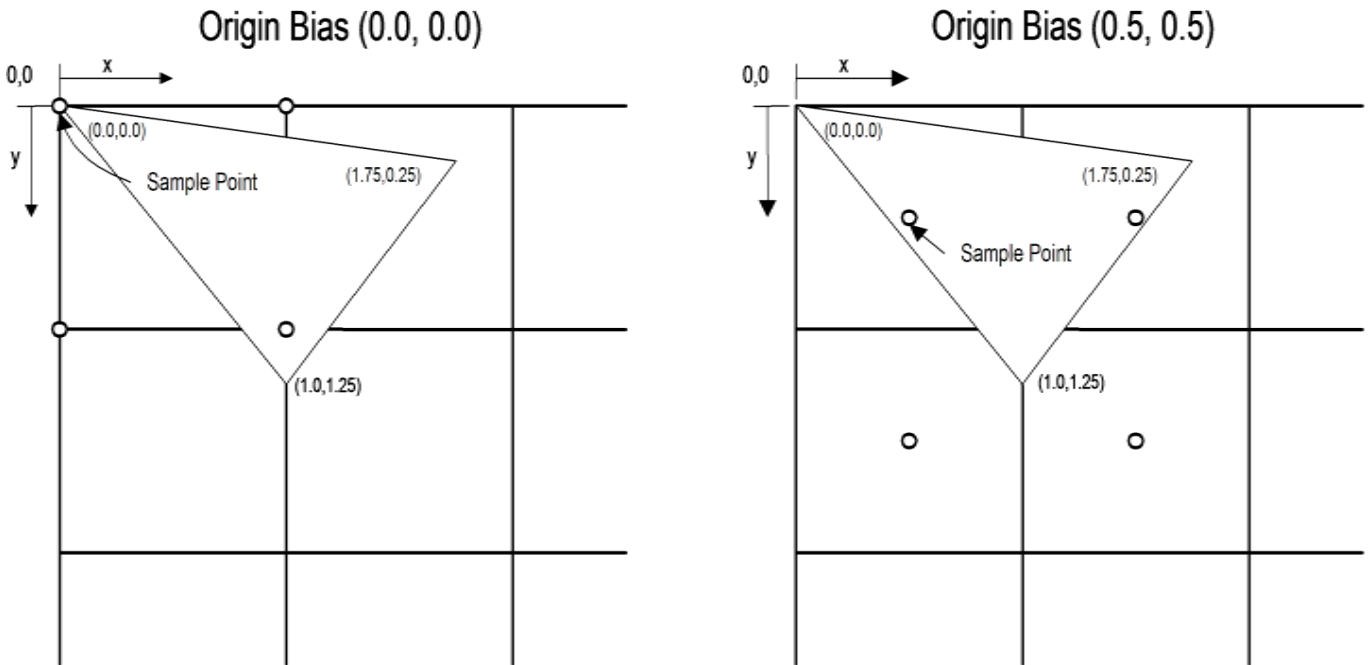
Destination Origin Bias

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.

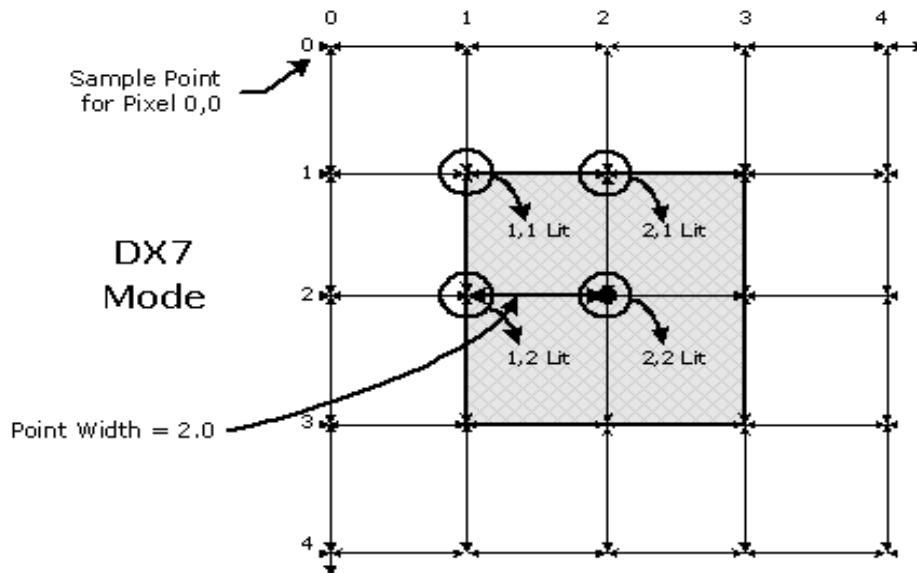
Destination Origin Bias



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



B 6846-01

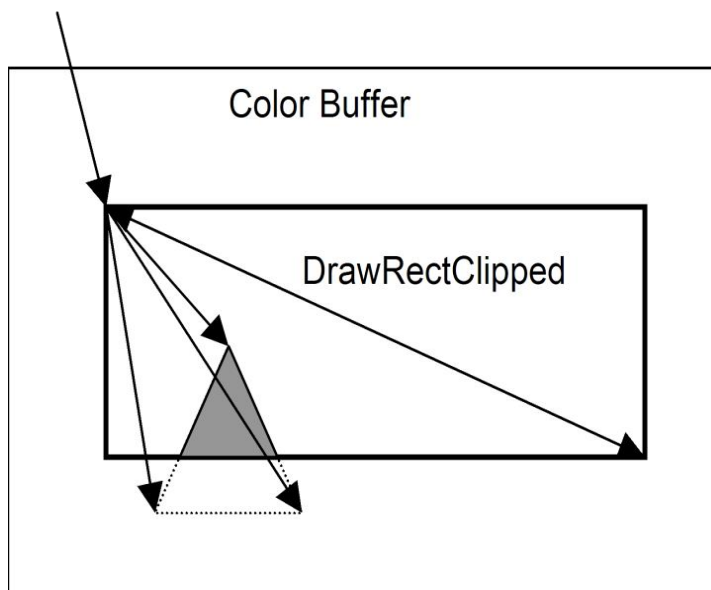
Drawing Rectangle Offset Application

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.

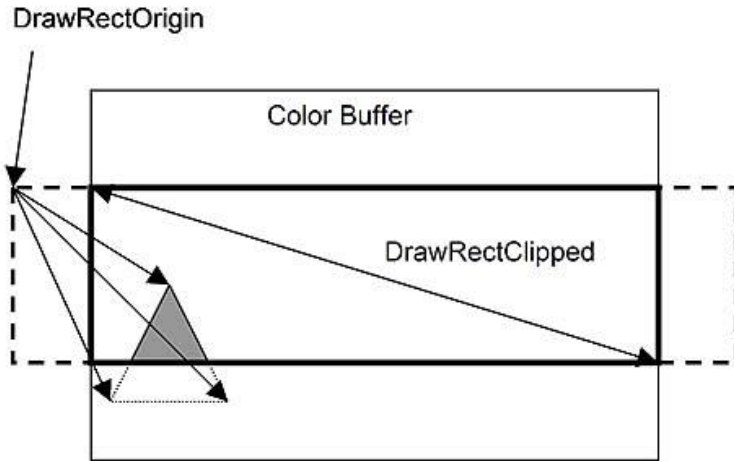
Clipping of objects which extend outside of the Drawing Rectangle occurs later in the pipeline. Note that this clipping is based on the "clipped" draw rectangle (as programmed via the **Clipped Drawing Rectangle** values in the 3DSTATE_DRAWING_RECTANGLE command), which must be clamped by software to the render target 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

DrawRectOrigin



Partially-offscreen Draw Rectangle



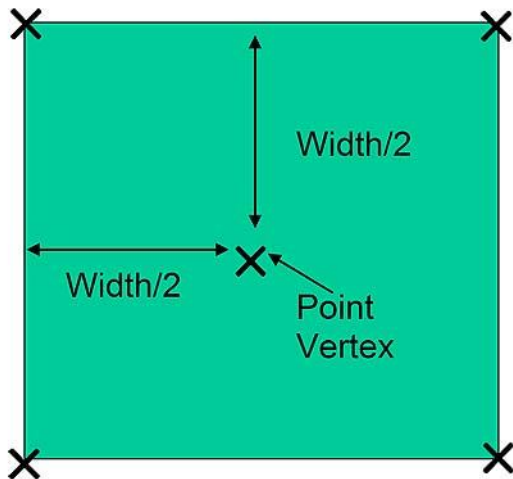
3DSTATE_DRAWING_RECTANGLE

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*.

Point Width Application

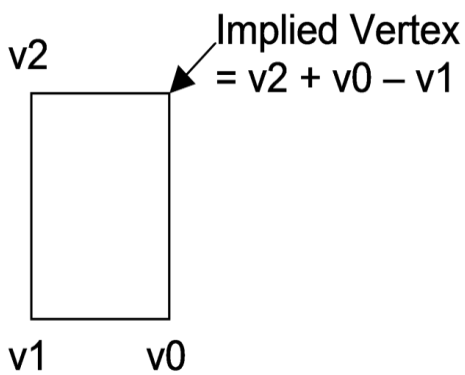


Z and W vertex attributes are copied from the single point center vertex to each of the four corner vertices.

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

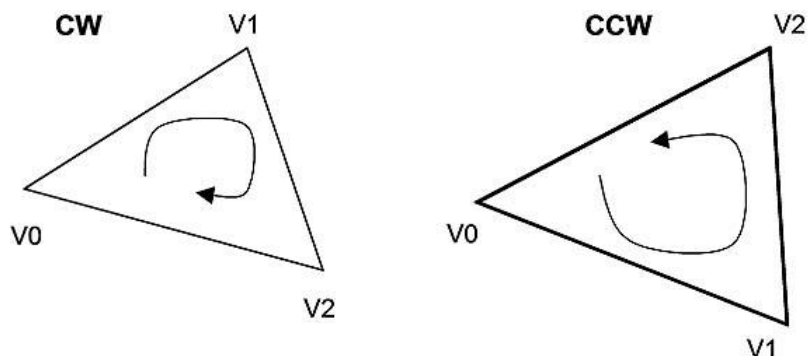
Triangle Orientation (Face) Culling

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 counterclockwise (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 counterclockwise (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 from 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.

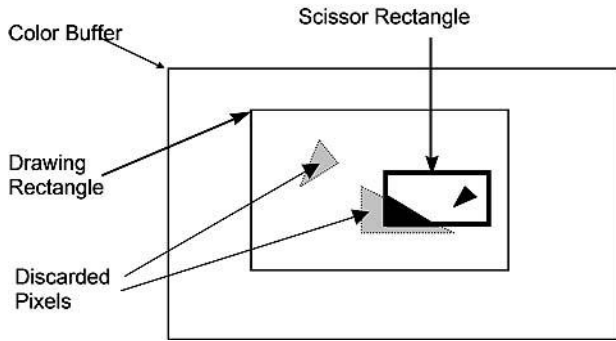
Cull Mode

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 $x_{min} > x_{max}$ or $y_{min} > y_{max}$ 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 coordinates 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 $x_{min} > x_{max}$ or $y_{min} > y_{max}$ 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.

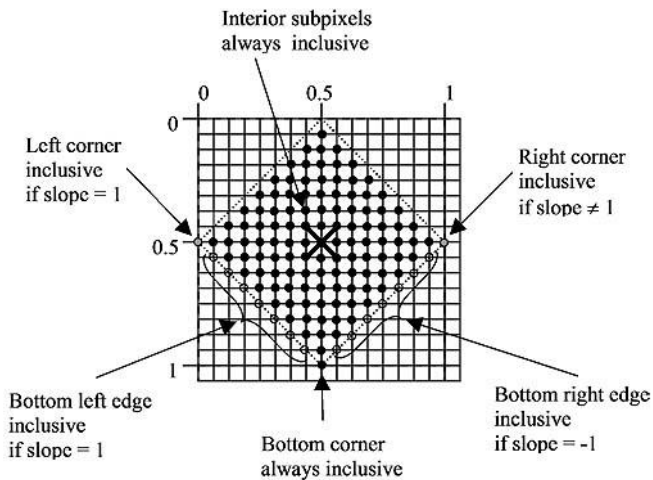
The rasterization rules also comply with the OpenGL conformance requirements (for 1-pixel wide non-smooth lines). Refer to the appropriate API specifications for details on these requirements.

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 $(1, +\infty]$ or $[-\infty, -1)$ 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)

```

```

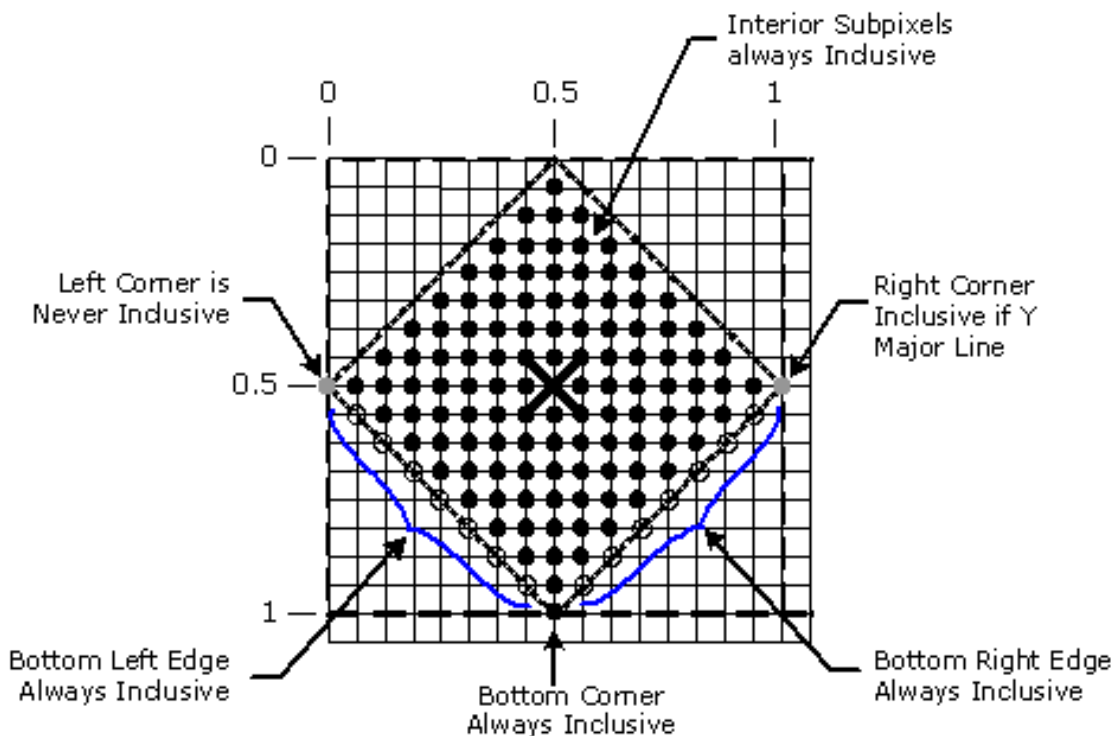
inside = interior || bottom_corner || (slopePosOne ? left_corner : right_corner) ||
(slopePosOne && left_edge) || (slopeNegOne && right_edge)

```

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.



B6849-01

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 $-1 \leq \text{slope} \leq 1$).** Including the right corner subpixel ensures that lines with slopes in the range ($>1, +\infty$] or $[-\infty, <-1$) 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)

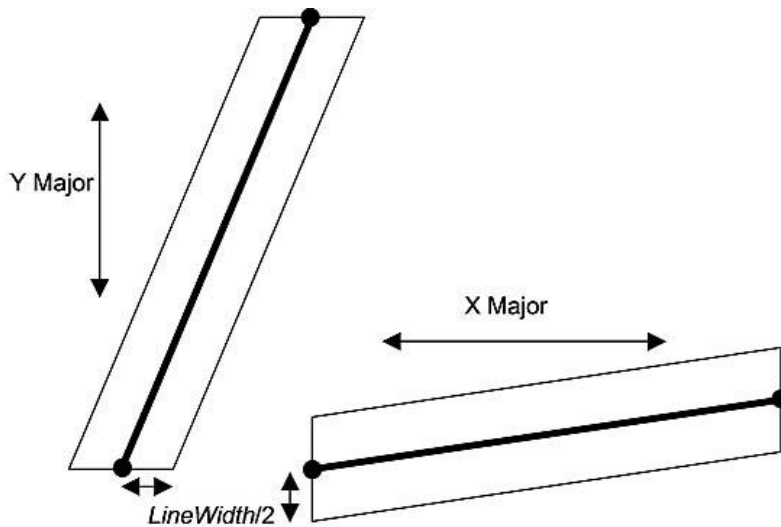
```

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 $\pm(\text{Line Width} / 2)$ in the direction of the minor axis, as shown in the following diagram.

Non-Antialiased Line Rasterization

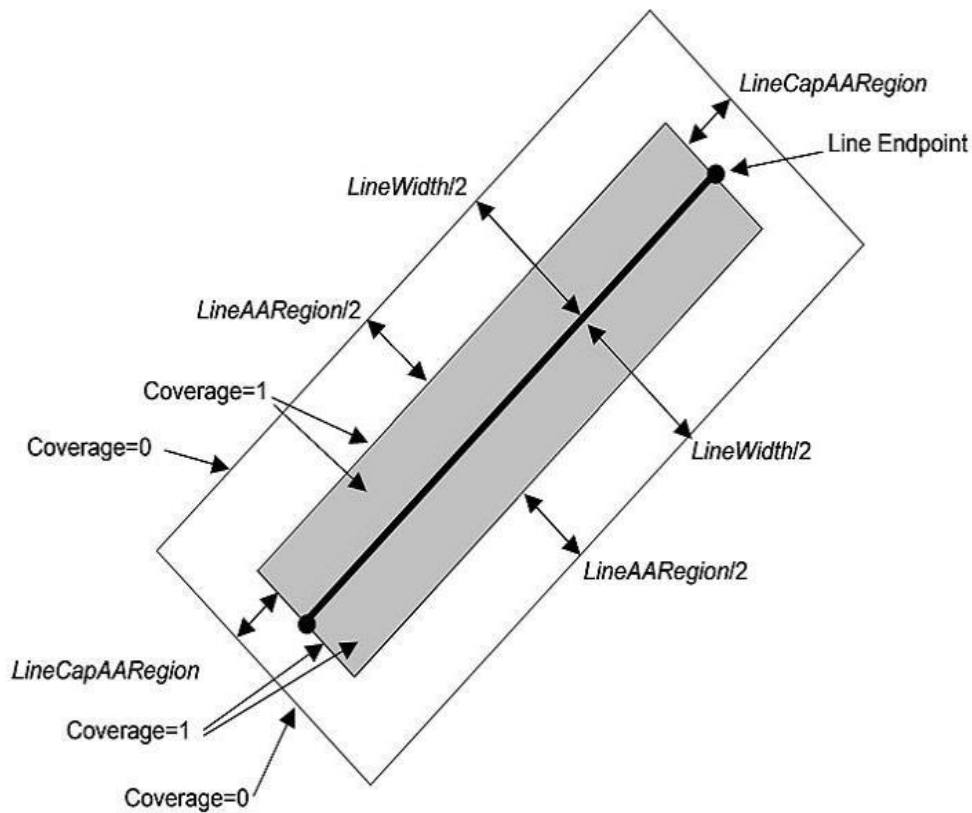


Anti-Aliased Line Rasterization

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 $(\text{LineWidth}/2)$ 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 its 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	$= (3DSTATE_RASTER:: \text{Global Depth Offset Enable Solid?} ((3DSTATE_RASTER::\text{Global Depth Offset Constant} \neq \text{IEEE_FP_ZERO}) \parallel (3DSTATE_RASTER::\text{Global Depth Offset Scale} \neq \text{IEEE_FP_ZERO})) : \text{Disable}$

Signal	SF_INT:: Global Depth Offset Enable Wireframe
Description	This field determines when Global Depth bias gets enabled.
Formula	$= (3DSTATE_RASTER:: \text{Global Depth Offset Enable Wireframe?} ((3DSTATE_RASTER::\text{Global Depth Offset Constant} \neq \text{IEEE_FP_ZERO}) \parallel (3DSTATE_RASTER::\text{Global Depth Offset Scale} \neq \text{IEEE_FP_ZERO})) : \text{Disable}$

Signal	SF_INT:: Global Depth Offset Enable Point
Description	This field determines when Global Depth bias gets enabled.
Formula	$= (3DSTATE_RASTER:: \text{Global Depth Offset Enable Point?} ((3DSTATE_RASTER::\text{Global Depth Offset Constant} \neq \text{IEEE_FP_ZERO}) \parallel (3DSTATE_RASTER::\text{Global Depth Offset Scale} \neq \text{IEEE_FP_ZERO})) : \text{Disable}$



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	$= (3DSTATE_RASTER::\ Global\ Depth\ Offset\ Enable\ Solid ?$ $($ $(3DSTATE_RASTER::Global\ Depth\ Offset\ Constant \neq IEEE_FP_ZERO) \ \ (3DSTATE_RASTER::Global\ Depth\ Offset\ Scale$ $\neq IEEE_FP_ZERO)$

); 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 back-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 specifies, 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	$(A1-A0)+1$ $(A1-A0) \leq -0.5$ $(A1-A0)-1$ $(A1-A0) \geq 0.5$ $(A1-A0)$ otherwise	$(A2-A0)+1$ $(A2-A0) \leq -0.5$ $(A2-A0)-1$ $(A2-A0) \geq 0.5$ $(A2-A0)$ 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

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.

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	$= (\text{WM_INT::WM_HZ_OP}) ?$ 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	$= (\text{3DSTATE_WM::ForceThreadDispatch} == \text{ON}) \parallel$ $\left(\text{3DSTATE_WM::ForceThreadDispatch} \neq \text{OFF} \right) \&\&$ $\&\& \text{WM_INT::WM_HZ_OP} \&\&$ $\&\& \text{3DSTATE_PS_EXTRA::PixelShaderValid} \&\&$ $\left(\text{!3DSTATE_PS_EXTRA::PixelShaderDoesNotWriteRT} \&\&$

	<pre> 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))) </pre>

Signal	WM_INT::Pixel Shader Computed Depth Mode
Description	This field specifies the computed depth mode for the pixel shader.
Formula	<pre> = (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 </pre>

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	MSRASTM E_OFF_PIXEL	MSRASTM E_OFF_PATTE R N	MSRASTM E_ON_PIXEL	MSRASTM E_ON_PATTE R N	*	*	*
WM_INT::WM_HZ_ OP	*	*	*	*	True	True	False
3DSTATE_WM_HZ_ OP:: Number of Multisamples	*	*	*	*	> NUMSAMPLES _1	NUMSAMPLES _1	*
WM_INT::Multisam ple Rasterization Mode	OFF_PIXEL	OFF_PATTE R N	ON_PIXEL	ON_PATTE R N	ON_PATTE R N	ON_PIXEL	Determined from Table 1 in 3D Pipeline Windower > Multisampli ng > Multisample Modes/Stat e)

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, <i>other than due to depth or stencil testing.</i>
Formula	<pre> = (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)) </pre>

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	<p>=</p> <p>(WM_INT::WM_HZ_OP ?</p> <p>Disable :</p> <p>(3DSTATE_RASTER::ForceSampleCount != NUMRASTSAMPLES_0) ?</p> <p>Enable :</p> <p>Disable</p>

Signal	WM_INT::Statistics Enable
Description	Enables Statistics
Formula	<p>= (WM_INT::WM_HZ_OP) ?</p> <p>Disable :</p> <p>3DSTATE_WM:: Statistics Enable</p>

Signal	WM_INT::Polygon Stipple Enable
Description	Enables Poly Stipple
Formula	<p>= (WM_INT::WM_HZ_OP) ?</p> <p>Disable :</p> <p>3DSTATE_WM::Polygon Stipple Enable</p>

Signal	WM_INT::WM_HZ_OP
Description	Enables WM_HZ_OP
Formula	<p>= (3DSTATE_WM_HZ_OP::DepthBufferClear </p> <p>3DSTATE_WM_HZ_OP::DepthBufferResolve </p> <p>3DSTATE_WM_HZ_OP::Hierarchical Depth Buffer Resolve Enable </p> <p>3DSTATE_WM_HZ_OP::StencilBufferResolve </p> <p>3DSTATE_WM_HZ_OP::StencilBufferClear </p> <p>3DSTATE_WM_HZ_OP::DepthBufferPartialResolve) ?</p> <p>Enable :</p> <p>Disable</p>

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	<pre>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; <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> Add this additional case: Case NUMSAMPLES_16: WM_INT:: Sample Mask Enable = 0xFFFF; break; </div> }</pre>

Signal	WM_INT:: Sample Mask
Description	Sets Sample Mask used in rasterization
Formula	<pre>= WM_INT:: Sample Mask Enable & (WM_INT::WM_HZ_OP) ? 3DSTATE_WM_HZ_OP::Sample Mask: 3DSTATE_SAMPLE_MASK::Sample Mask)</pre>

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
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 ?	Disable	Disable

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable (full or partial)	Hierarchical Depth Buffer Resolve Enable
		Enable : 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 Stencil Enable	Disable	Disable	Disable	Disable
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::Sa mple Mask	3DSTATE_WM_HZ_OP::Sa mple Mask	3DSTATE_WM_HZ_OP::Sa mple Mask	3DSTATE_WM_HZ_OP::Sa mple Mask

State	Stencil buffer Clear	Depth buffer clear	Depth Buffer Resolve Enable (full or partial)	Hierarchical Depth Buffer Resolve Enable
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	<pre>= (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))</pre>

Signal	WM_INT::DepthTestFunction
Description	Depth Test Function
Formula	<pre>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)</pre>

Signal	WM_INT::DBT_enable
Description	Depth Bounds Test enable
Formula	<pre>(WM_INT::WM_HZ_OP) ? 0 : 3DSTATE_DEPTH_BOUNDS::DBT_enable && (3DSTATE_DEPTH_BUFFER::SURFACE_TYPE != NULL)</pre>

Signal	WM_INT::StencilWriteEnable
Description	Enables writes to the Stencil Buffer
Formula	<pre> = 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)) </pre>

Signal	WM_INT::StencilWriteEnable
Description	Enables writes to the Stencil Buffer
Formula	<pre> = 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)) </pre>

Signal	WM_INT::DepthWriteEnable
Description	Enables Depth Write
Formula	<pre> = (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) </pre>

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

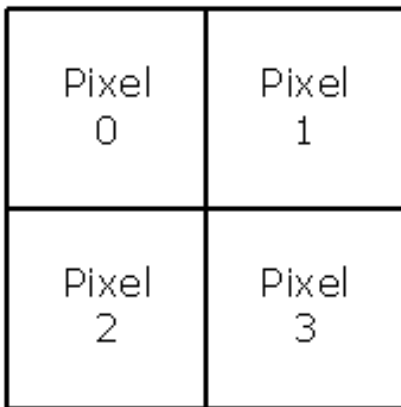
Description	Stencil Write Mask
Formula	= 3DSTATE_WM_HZ_OP::StencilClear ? 0xFF : 3DSTATE_WM_DEPTH_STENCIL::Backface Stencil Write Mask

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:

$$\text{edge0alpha} * \text{edge1alpha} = 148/255 = 0.580392157$$

Since $\text{edge0alpha} = \text{edge3alpha}$ we get:

$$(\text{edge0alpha})^2 = 0.580392157$$

$$\text{edge0alpha} = \sqrt{0.580392157} = 0.761834731 \text{ at the center pixel}$$

$$\text{The desired alpha for pixel 1} = 54/255 = 0.211764706$$

$$\text{The slope is } (0.761834731 - 0.211764706) = 0.550070025$$

Since we are using 8 bit precision, the slope becomes

$$\text{AA Coverage [EndCap] Slope} = 0.55078125$$

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 $1.0f / \text{Line Stipple Repeat Count}$ 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.

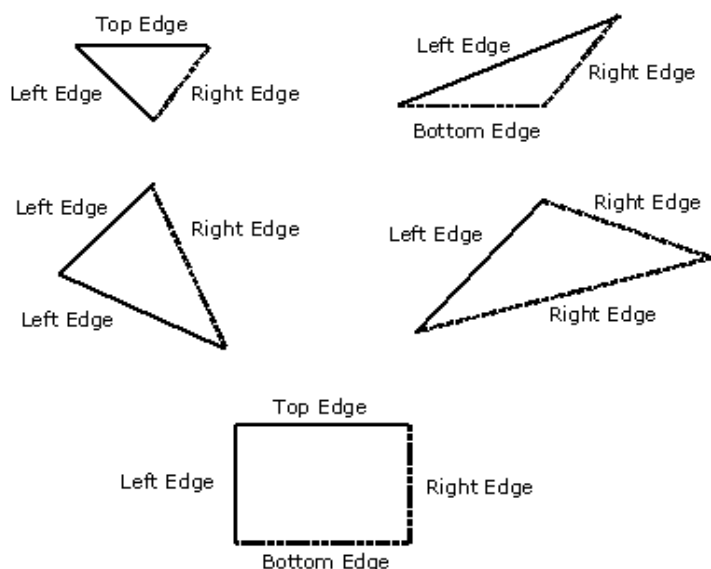
3DSTATE_LINE_STIPPLE

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



B.6851-01

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	Enable Render Target Independent Rasterization.

State Element	Source	Description
	!= NUMRASTSAMPLES_0	
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.
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	NUMRASTSAMPLES_1	NUMRASTSAMPLES_1	> NUMRASTSAMPLES_1	> NUMRASTSAMPLES_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

Number of Multisamples	MS RAST MODE	MS DISP MODE	HW Mode
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
		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

Number of Multisamples	MS RAST MODE	MS DISP MODE	HW Mode
			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 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

Rast Number of Samples	MS RAST MODE	HW Mode
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 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.

PixelPipe Hashing

Wherever hashing to slice is mentioned, it refers to PixelPipe. For example, if a configuration has two PixelPipes and 2 DS for each PixelPipe, the mapping is described below for the purpose of the pixel distribution.

- PixelPipe0 = LogicalSlice0 = DSS0 & DSS1
- PixelPipe1 = LogicalSlice1 = DSS2 & DSS3

SubSlice Hashing

2-way Hashing

$\text{subslice_id} = X[0] \wedge Y[0]$

3-way Hashing

If $X+Y$ is divisible by 3, then $\text{subslice_id}=0$, else $\text{subslice_id} = 1 + X[2] \wedge Y[0]$

	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.

`slice_id = slice_hash_table[Y[3:0]][X[3:0]]`

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 active slices, and the following effective `slice_id`.

`slice_id = (X[3:0] + Y[3:0]) % active_number_slices`

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**. First entry in the array is `slices=2`. No `slice_hash_table` is fetched when there is only one active slice.



here.

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

$$\text{Hz Slice} = \text{Slice Hash Table}[Y][X] \% \text{NUM_HZ_IN_A_SLICE}$$

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



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_sliceen is: 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 here:

	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.

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**. First entry in the array is slices==2. No slice_hash_table is fetched when there is only one active slice.

3DSTATE_SLICE_TABLE_STATE_POINTERS

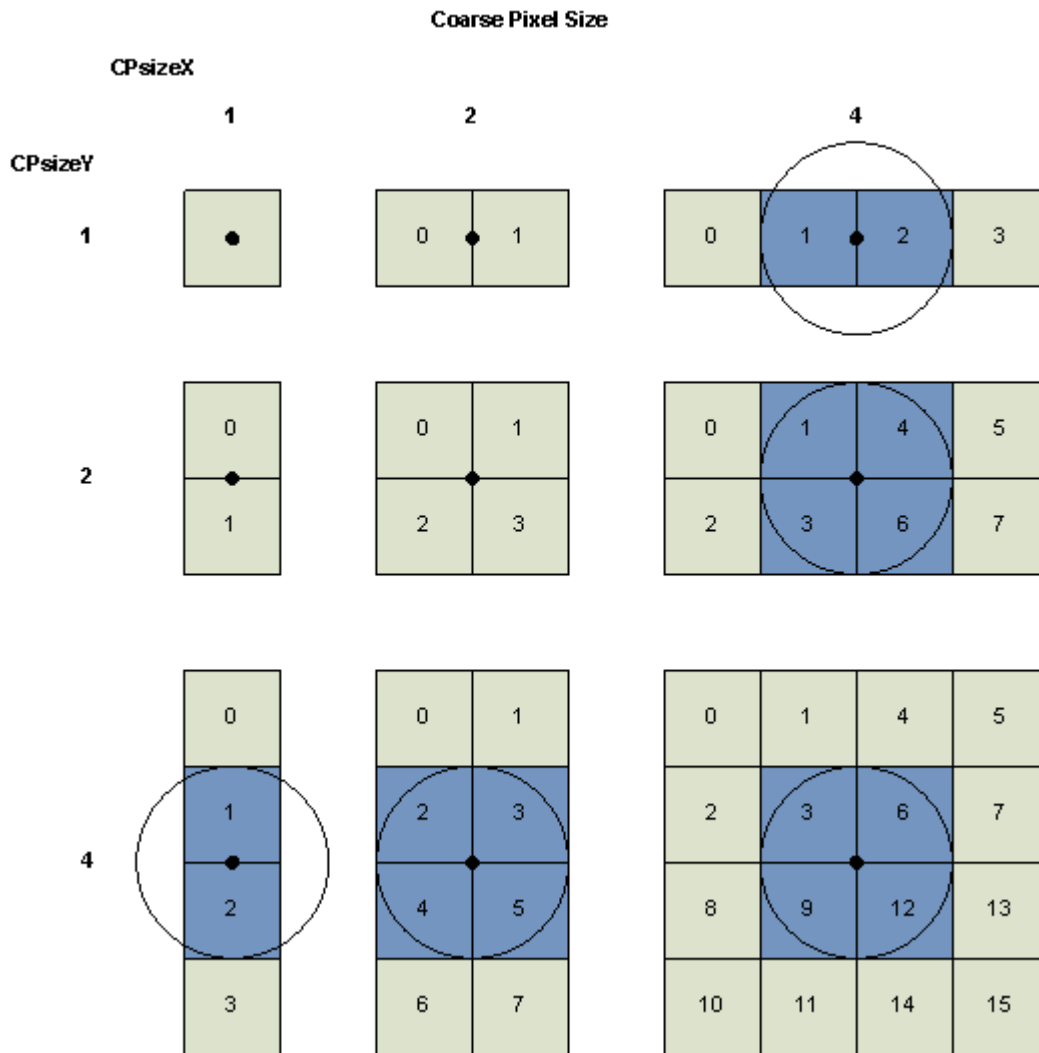
SLICE_HASH_TABLE - SLICE_HASH_TABLE

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 CP sizes 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 (C_x, C_y) be the center position, in pixel coordinates, specified as `CoarsePixelSizeState.CenterX` and `CoarsePixelSizeState.CenterY`. These values must be in $[0.0f, \text{MAX_RENDER_TARGET_LINEAR_RESOLUTION}]$.
- 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 $[0.0f, \text{MAX_RENDER_TARGET_LINEAR_RESOLUTION} * \text{sqrt}(2.0f)]$ (the minimum and maximum radii in pixel coordinates).
- Let R_{\max} be the maximum radius specified as `CoarsePixelSizeState.RadiusMaxSize`. These values must be in $[0.0f, \text{MAX_RENDER_TARGET_LINEAR_RESOLUTION} * \text{sqrt}(2.0f)]$ (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:

$$\text{DeltaX} = (C_x - P_x)$$

$$\text{DeltaY} = (C_y - P_y)$$

$$\text{If } (A_{\text{ratio}} \leq 1) \text{ DeltaY} = A_{\text{ratio}} * \text{DeltaY}$$

$$\text{Else DeltaX} = (1/A_{\text{ratio}}) * \text{DeltaX}$$

$$D = \text{Distance}(\text{DeltaX}, \text{DeltaY})$$

$$\text{CoarsePixelSize}_x = M_x * (D - R_{\min}) + S_x^{\min}$$

$$\text{CoarsePixelSize}_y = M_y * (D - R_{\min}) + S_y^{\min}$$

$$\text{CoarsePixelSize}_x = \text{clamp}(\text{CoarsePixelSize}_x, S_x^{\min}, S_x^{\max})$$

$$\text{CoarsePixelSize}_y = \text{clamp}(\text{CoarsePixelSize}_y, S_y^{\min}, S_y^{\max})$$

$$\text{RequestedCoarsePixelSize} = (\text{CoarsePixelSize}_x, \text{Coars}, e\text{PixelSize}_y)$$

Where `Distance()` is the Pythagorean distance function between two points defined by the following function:

$$Distance(\Delta X, \Delta Y) = \sqrt{\Delta X^2 + \Delta Y^2}$$

$$Distance(\Delta X, \Delta Y) = \sqrt{\Delta X^2 + \Delta Y^2}$$

However, because this function may be expensive to compute exactly, the distance function may be approximated by HW.

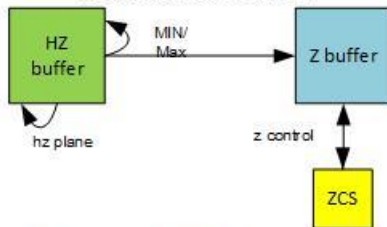
Early Depth/Stencil Processing

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.

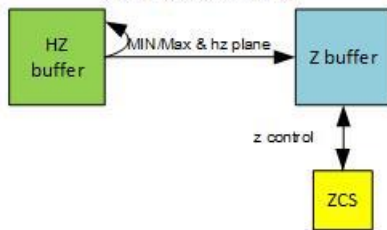
Compressed Depth Buffer

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.

Hiz buffer enable, Hiz buffer non-write-thru
(Gen12 non-texture mode)



Hiz buffer enable, Hiz buffer write-thru
(Gen12 texture mode)



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. This buffer is supported only in Tile Y memory.

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	$\text{ceiling}(Z_Width / 16) * 16$	$\text{ceiling}((HZ_QPitch/2)/16) * 16 * Z_Depth$	see below
SURFTYPE_3D			not applicable
SURFTYPE_CUBE	$\text{ceiling}(Z_Width / 16) * 16$	$\text{ceiling}((HZ_QPitch/2)/16) * 16 * 6 * Z_Depth$	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:

$$HZ_QPitch = h_0 + \max \left(h_1, \sum_{i=2}^m h_i \right)$$

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 next 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 through 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.
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 next 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
<p>Note: The PS Unit also receives 3DSTATE_PS_BLEND, 3DSTATE_SAMPLEMASK, 3DSTATE_MULTISAMPLE, and 3DSTATE_PS_EXTRA.</p>	

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)

	<pre>(3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_SRC1_ALPHA) (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_INV_SRC1_COLOR) (3DSTATE_PS_BLEND::DestinationAlphaBlendFactor == BLENDFACTOR_INV_SRC1_ALPHA)</pre>
--	--

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.

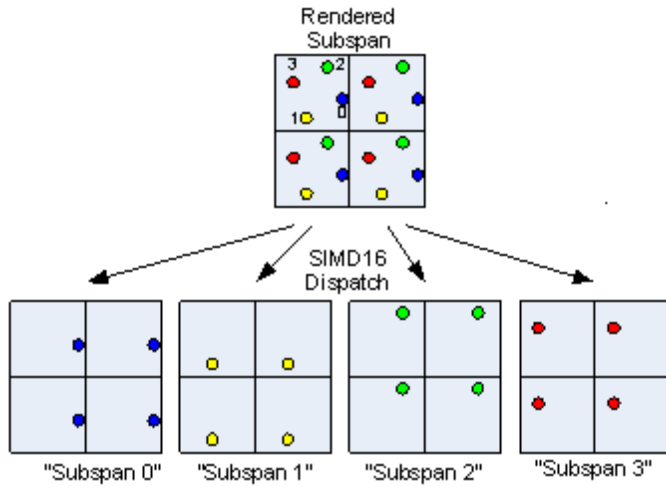
Programming Note	
Context:	MSDISPMODE_PERPIXEL Thread Dispatch
When NUM_MULTISAMPLES == 16 (i.e. 16x MSAA) and PS_DISPATCH_MODE is PER_PIXEL, SIMD32 pixel shader is not supported.	

MSDISPMODE_PERSAMPLE Thread Dispatch

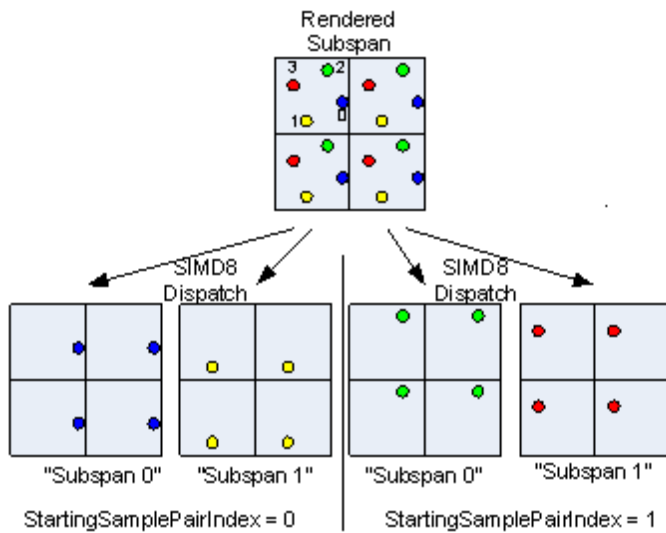
In PERSAMPLE mode, the pixel shader needs to operate on a sample vs. pixel basis (although this collapses in NUMSAMPLES_1 mode) Instead of processing strictly different subspans in parallel, the PS kernel processes different sample indices of one or more subspans in parallel For example, a SIMD16 dispatch 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 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	$Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]$ $Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]$ $Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0]$ $Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]$ $Slot[19:16] = Subspan[4].Pixel[3:0].Sample[0]$ $Slot[23:20] = Subspan[5].Pixel[3:0].Sample[0]$ $Slot[27:24] = Subspan[6].Pixel[3:0].Sample[0]$ $Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]$
	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	$Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]$ $Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1]$ $Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2]$

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		$Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]$ $Slot[19:16] = Subspan[0].Pixel[3:0].Sample[4]$ $Slot[23:20] = Subspan[0].Pixel[3:0].Sample[5]$ $Slot[27:24] = Subspan[0].Pixel[3:0].Sample[6]$ $Slot[31:28] = Subspan[0].Pixel[3:0].Sample[7]$
	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	$Dispatch[i]: (i=0, 2)$ $SSPI = i$

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
		$Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]$ $Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]$ $Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2]$ $Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]$
	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.

This payload does not apply to the contiguous dispatch modes. The payload for these modes is documented in the section titled *PS Thread Payload for Contiguous Dispatch*.

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 Pointer: Specifies the 1K-byte aligned pointer to the scratch space available for this PS fused thread. This is specified as an offset to the General State Base Address . Format = GeneralStateOffset[31:10]
	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

DWord	Bits	Description
		specified as an offset from the Dynamic State Base Address . Format = DynamicStateOffset[31:5]
	4	Reserved
	3:0	<p>Per Thread Scratch Space: Specifies the amount of scratch space allowed to be used by this fused thread.</p> <p>Programming Notes: This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it.</p> <p>Format = U4</p> <p>Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two</p>
R0.2	31:0	Reserved: Delivered as zeros (reserved for message header fields).
R0.1	31:6	<p>Color Calculator State Pointer: Specifies the 64-byte aligned pointer to the Color Calculator 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.</p> <p>Format = DynamicStateOffset[31:5]</p>
	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	<p>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).</p> <p>Format: (See 3DPRIMITIVE command in <i>3D Pipeline</i>.)</p>
R1.7	31:16	<p>Pixel/Sample Mask (SubSpan[3:0]): Indicates which pixels within the four subspans are lit.</p> <p>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.</p> <p>This field must not be modified by the Pixel Shader kernel.</p>
	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

DWord	Bits	Description														
		kill instructions.														
R1.6	31	Reserved														
	30:27	<p>Viewport Index 1: Specifies the index of the viewport currently being used for upper SIMD8 of dual-SIMD8 thread.</p> <p>Format = U4</p> <p>Range = [0,15]</p>														
	26:16	<p>Render Target Array Index 1: Specifies the array index to be used for upper SIMD8 of dual-SIMD8 thread</p> <p>Following surface types:</p> <p>SURFTYPE_1D: specifies the array index Range = [0,2047]</p> <p>SURFTYPE_2D: specifies the array index Range = [0,2047]</p> <p>SURFTYPE_3D: specifies the "r" coordinate Range = [0,2047]</p> <p>SURFTYPE_CUBE: specifies the face identifier Range = [0,5]</p> <table border="1" data-bbox="407 947 824 1262"> <thead> <tr> <th>Face</th> <th>Render Target Array Index</th> </tr> </thead> <tbody> <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> </tbody> </table> <p>Format = U11</p>	Face	Render Target Array Index	+x	0	-x	1	+y	2	-y	3	+z	4	-z	5
	Face	Render Target Array Index														
	+x	0														
	-x	1														
	+y	2														
	-y	3														
+z	4															
-z	5															
15	<p>Front/Back Facing Polygon 1: Determines whether the polygon 1 is front or back facing.</p> <p>Used by the render cache to determine which stencil test state to use.</p> <p>0: Front Facing</p> <p>1: Back Facing</p>															
14:9	Reserved															
8	Reserved for expansion of Starting Sample Pair Index 1															
7:6	<p>Starting Sample Pair Index 1: Indicates the index of the second sample pair of dual-SIMD8 dispatch.</p> <p>Format = U2</p> <p>Range = [0,3]</p>															
5:0	Reserved															

DWord	Bits	Description			
R1.5	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			
R1.4	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			
R1.3	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			
R1.2	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			
R1.1	31	Reserved			
	30:27	Viewport Index 0: Specifies the index of the viewport currently being used. Format = U4 Range = [0,15]			
	26:16	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 border="1" data-bbox="406 1816 824 1913"> <thead> <tr> <th>Face</th> <th>Render Target Array Index</th> </tr> </thead> <tbody> <tr> <td>+x</td> <td>0</td> </tr> </tbody> </table>	Face	Render Target Array Index	+x
Face	Render Target Array Index				
+x	0				

DWord	Bits	Description										
		<table border="1"> <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> <p>Format = U11</p>	-x	1	+y	2	-y	3	+z	4	-z	5
-x	1											
+y	2											
-y	3											
+z	4											
-z	5											
	15	<p>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.</p> <p>0: Front Facing 1: Back Facing</p>										
	14:9	Reserved										
	8	Reserved for expansion of Starting Sample Pair Index 0										
	7:6	<p>Starting Sample Pair Index 0: Indicates the index of the first sample pair of the dispatch.</p> <p>Format = U2 Range = [0,3]</p>										
	5:0	Reserved										
R1.0	31:20	Reserved										
	19:16	<p>MSAA rate (multisample count)</p> <p>Format: U4 [1..16]</p> <p>This field specifies MSAA sampling rate (required for PS+S monolithic shader).</p>										
	15:8	<p>ActualCoarsePixelShadingSize.Y if coarse pixel dispatch</p> <p>Format: U8</p> <p>This field specifies size (in pixels) of coarse pixel shading rate in Y dimension. Valid values are 1, 2, and 4.</p> <p>Note: coarse shading rate is constant for all coarse pixels in same thread dispatch.</p>										
	15:12	<p>Slot 3 SampleID (if pixel or sample dispatch)</p> <p>Format = U4</p> <p>1X MSAA range: [0]</p> <p>2X MSAA range [0,1]</p> <p>4X MSAA range [0..3]</p> <p>8X MSAA range [0..7]</p>										

DWord	Bits	Description
	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.
	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 <i>32-pixel dispatch</i> .
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.

DWord	Bits	Description
	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
	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

DWord	Bits	Description
		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] 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 <i>8-pixel dispatch</i> 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

DWord	Bits	Description
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 (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 <i>8-pixel dispatch</i> .
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

DWord	Bits	Description
		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 <i>8-pixel dispatch</i> .
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
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

DWord	Bits	Description
	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. 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

DWord	Bits	Description
		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 <i>8-pixel dispatch</i> .
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.
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.

DWord	Bits	Description
		R34-R57: Delivered only if the corresponding Barycentric Interpolation Mode bit is set and this is a <i>32-pixel dispatch</i> .
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
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 <i>32-pixel dispatch</i> .
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.

DWord	Bits	Description
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
		R60-R61: Delivered only if Pixel Shader Uses Source W is set and this is a <i>32-pixel dispatch</i> .
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

DWord	Bits	Description
		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
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

DWord	Bits	Description
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) 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.

DWord	Bits	Description
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
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

DWord	Bits	Description
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
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
R70.5	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

DWord	Bits	Description
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: U0.4 (Upper two bits MBZ.) Subpixel Y offset of Sample 15 relative to the UL pixel origin Range: [0,0.9375] (0/16 - 15/16) The number of valid sub-sample offsets is the number of MSAA samples
	23:22	Reserved – MBZ
	21:16	Sub-sample Y offset for sample 14
	15:14	Reserved – MBZ
	13:8	Sub-sample Y offset for sample 13
	7:6	Reserved – MBZ
	5:0	Sub-sample Y offset for sample 12
R72.6	31:30	Reserved – MBZ
	29:24	Sub-sample Y offset for sample 11
	23:22	Reserved – MBZ
	21:16	Sub-sample Y offset for sample 10
	15:14	Reserved – MBZ
	13:8	Sub-sample Y offset for sample 9
	7:6	Reserved – MBZ
	5:0	Sub-sample Y offset for sample 8
R72.5	31:30	Reserved – MBZ
	29:24	Sub-sample Y offset for sample 7
	23:22	Reserved – MBZ
	21:16	Sub-sample Y offset for sample 6
	15:14	Reserved – MBZ

DWord	Bits	Description
	13:8	Sub-sample Y offset for sample 5
	7:6	Reserved – MBZ
	5:0	Sub-sample Y offset for sample 4
R72.4	31:30	Reserved – MBZ
	29:24	Sub-sample Y offset for sample 3
	23:22	Reserved – MBZ
	21:16	Sub-sample Y offset for sample 2
	15:14	Reserved – MBZ
	13:8	Sub-sample Y offset for sample 1
	7:6	Reserved – MBZ
	5:0	Sub-sample Y offset for sample 0
R72.3	31:30	Reserved – MBZ
	29:24	Sub-sample X offset for sample 15 Format: U0.4 (Upper two bits MBZ.) Subpixel X offset of Sample 15 relative to the UL pixel origin Range: [0,0.9375] (0/16 - 15/16) The number of valid sub-sample offsets is the number of MSAA samples
	23:22	Reserved – MBZ
	21:16	Sub-sample X offset for sample 14
	15:14	Reserved – MBZ
	13:8	Sub-sample X offset for sample 13
	7:6	Reserved – MBZ
	5:0	Sub-sample X offset for sample 12
	R72.2	31:30
29:24		Sub-sample X offset for sample 11
23:22		Reserved – MBZ
21:16		Sub-sample X offset for sample 10
15:14		Reserved – MBZ
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

DWord	Bits	Description
	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	<p>Primitive Full Pixel Coverage bitmask</p> <p>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.</p> <p>Format: U32</p>
R73.4:6	31:0	Reserved - MBZ
R73.3	31:24	Reserved - MBZ
	23:0	<p>Primitive Ymax - maximum value of sub-pixel aligned position.Y attribute in screen-space, computed from vertices forming an input primitive.</p> <p>Format: U16.8</p>
R73.2	31:24	Reserved - MBZ
	23:0	<p>Primitive Ymin - minimum value of sub-pixel aligned position.Y attribute in screen-space, computed from vertices forming an input primitive</p> <p>Format: U16.8</p>
R73.1	31:24	Reserved - MBZ
	23:0	<p>Primitive Xmax - maximum value of sub-pixel aligned position.X attribute in screen-space, computed from vertices forming an input primitive</p> <p>Format: U16.8</p>
R73.0	31:24	Reserved - MBZ
	23:0	<p>Primitive Xmin - minimum value of sub-pixel aligned position.X attribute in screen-space, computed from vertices forming an input primitive</p> <p>Format: U16.8</p>

DWord	Bits	Description
R74		Defines sub-pixel aligned quad and full pixel coverage mask for poly 1.
		<p>Optional Padding before the Start of Constant/Setup Data</p> <p>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).</p>
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	<p>Constant Data (optional):</p> <p>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.</p>
		<p>Setup Data (Attribute Vertex Deltas) for dispatches other than dual 8-pixel dispatch.</p> <p>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. Thus, the total number of registers sent is equal to 2 * Number of Output Attributes.</p>
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

DWord	Bits	Description
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 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.
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 is supported and 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 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
<ul style="list-style-type: none"> • 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 <u>not</u> 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
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



Blending	Pre-Blend Color Clamp	Clamp 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
<ul style="list-style-type: none"> 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
<ul style="list-style-type: none"> 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 <i>Sampling Engine</i>. 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:

$$\begin{aligned} \text{src_term} &= \text{src_blend_factor} * \text{src_color} \\ \text{dst_term} &= \text{dst_blend_factor} * \text{dst_color} \\ \text{color output} &= \text{blend_function}(\text{src_term}, \text{dst_term}) \end{aligned}$$

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 = 2 nd output from PS in Dual-Source 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
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)

Blend Function	Operation (for each color component)
	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.

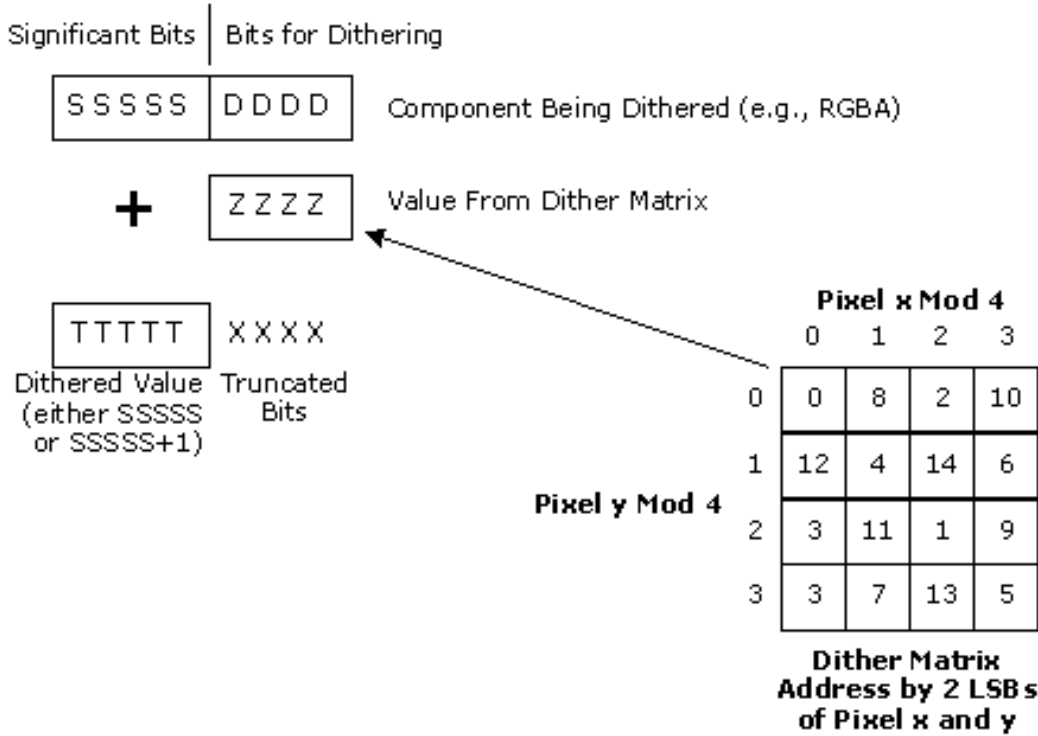
Regardless of the setting of **Post-Blend Color Clamp Enable**, when Blending is enabled color components will be automatically clamped to (at least) the rendertarget surface format range at this stage of the pipeline.

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.



B.6852-01

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 <i>StencilTestReference</i> 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 $a^{(2.4)}$ 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 does not exist. It 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 does not exist. It 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

This section of the BSpec discusses the programming the General Purpose GPU Pipeline, including:

- how GPGPU and Media threads are dispatched,
- thread resource management,
- how thread groups are tracked, and
- preemption and context switching.

General Purpose Compute Model

The General Purpose (GP) compute model consists of a host connected to one or more compute devices. Each compute device consists of several Processing Elements (PEs), also known as Execution Units (EUs). Applications are then architected as a combination of host software (per the host framework), and kernels that are submitted by the host to run on the PEs with a pre-defined decoupling point.

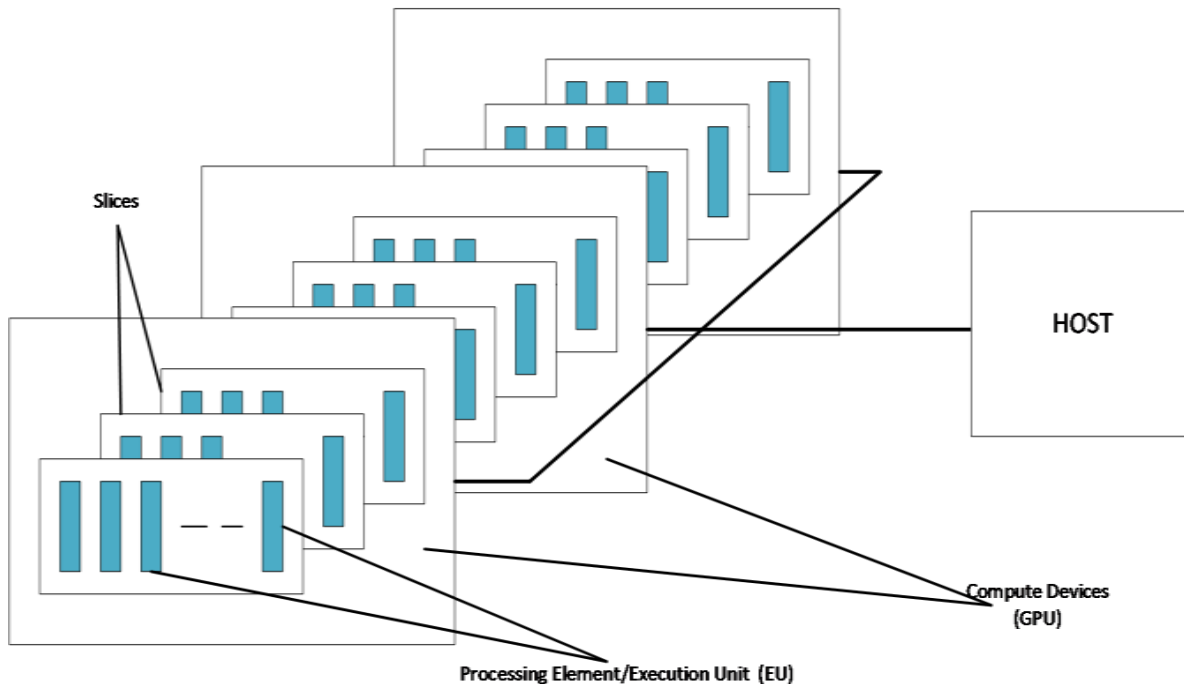


Figure: General Purpose Compute Model

The GP compute architecture contains two distinct units of execution: a host program and a set of kernels that execute within the context set by the host. The host interacts with these kernels through a command queue. Each device may have its own command queue. When a command is submitted into the command queue, the command is checked for dependencies and then executed on a PE inside the compute unit clusters. Once the command has finished executing, the kernel communicates an end of life cycle through "end of thread" message.

The GP execution model determines how to schedule and execute the kernels. When a kernel-enqueue command submits a kernel for execution, the command defines an index space or N-dimensional range. A kernel-instance consists of the kernel, the argument values associated with the kernel, and the parameters that define the index space. When a compute device executes a kernel-instance, the kernel function executes for each point in the defined index space or N-dimensional range.

An executing kernel function is called a work-item, and a collection of these work-items is called a work-group. A compute device manages work-items using work-groups. Individual work-items are identified by either a global ID, or a combination of the work-group ID and a local ID inside the work-group.

The work-group concept, which essentially runs the same kernel on several unit items in a group, captures the essence of data parallel computing. The PEs can organize work-items in vector format and run the same kernel on the vector, hence speeding up the compute for all such applications.

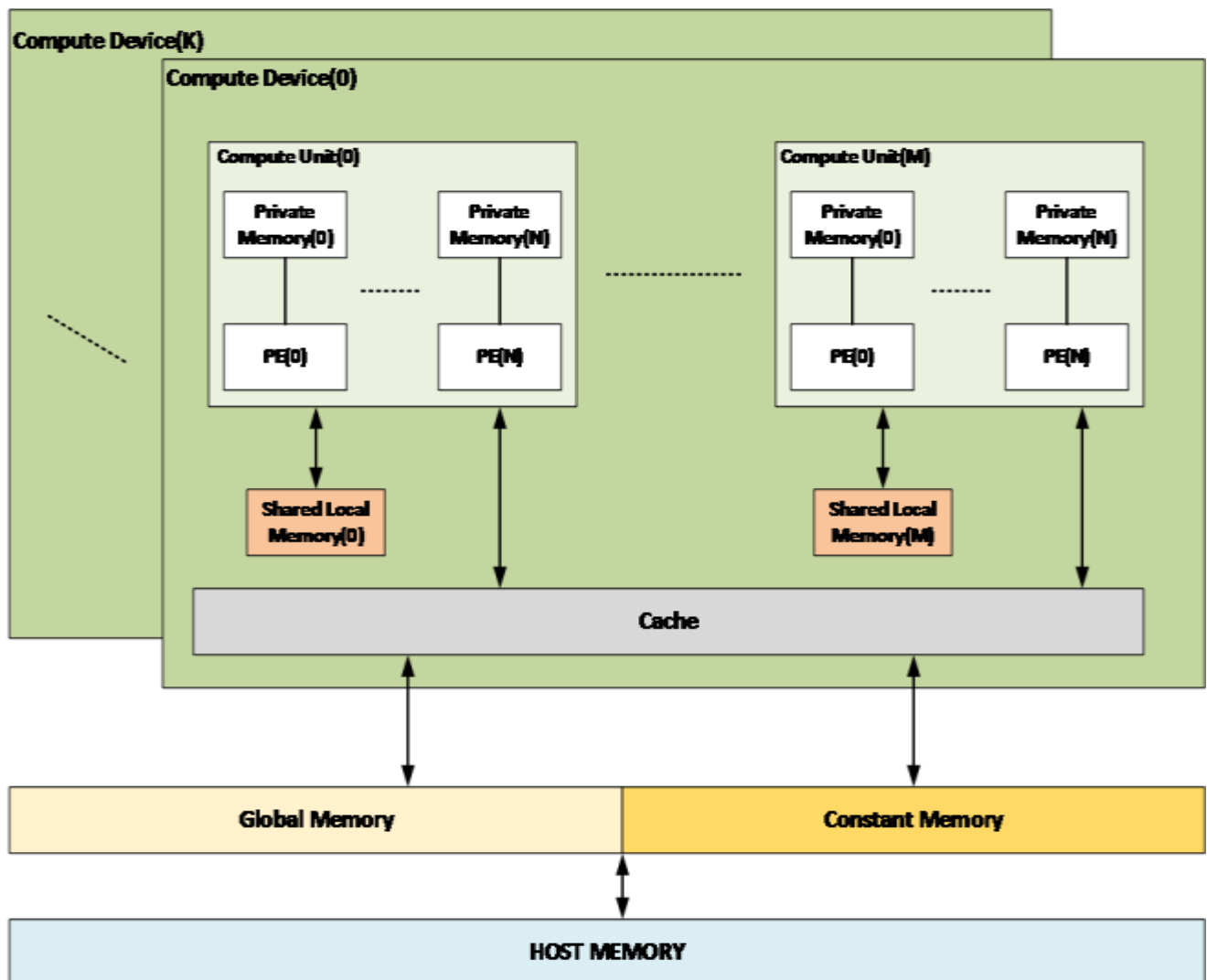
A device can compute each work-group in any arbitrary order. Also, the work-items within a single work-group execute concurrently, with no guarantee on the order of progress. A high level work-group function, like Barriers, applies to each work-item in a work-group, to facilitate the required synchronization points. Such a work-group function must be defined so that all work-items in the work-group encounter precisely the same work-group function.

Synchronization can also occur at the command level, where the synchronization can happen between commands in host command-queues. In this mode, one command can depend on execution points in another command or multiple commands.

Other types of synchronization based on memory-order constraints inside a program include Atomics and Fences. These synchronization types control how a memory operation of any particular work-item is made visible to another, which offers micro-level synchronization points in the data-parallel compute model.

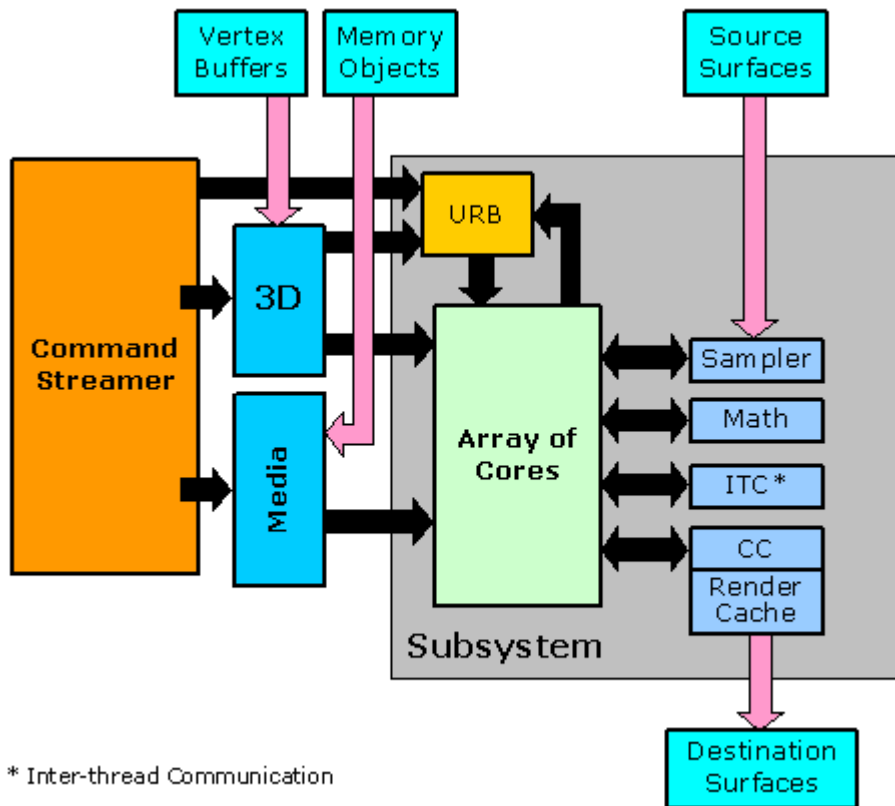
The memory model for a general-purpose engine is partitioned into host-side memory and device-side memory, using Shared Virtual Memory (SVM) to move objects between the two sides. The device-side memory is further divided into four distinct disjoint regions as defined below:

- Global Memory: Memory region accessible to all work-items within a context.
- Constant Memory: Memory region that remains constant during kernel execution.
- Local Memory: Memory region local and exclusive to a work-group (also called SLM).
- Private memory: Memory region private to a work-item.



GPGPU Context in GPU Hardware

The following block diagram provides an overview of the GPU Hardware. The media pipeline and the 3D fixed function pipeline are positioned in parallel. Concurrency of the media pipeline and the 3D pipeline is not supported. At any given time, only one of the two pipelines can be activated. A programmer can switch between the two pipelines within a single context by using the MI_PIPELINE_SELECT command.



* Inter-thread Communication

B.6676-01

The media pipeline provides media functions and has media-specific fixed function capability. These fixed functions are able to control shared functions and resources, feed generic threads to the Execution Units (EUs), and interact with generic threads during runtime. Since a programmer can use the media pipeline for non-media applications, the media pipeline can also be called the general purpose pipeline. Henceforth, we refer to this fixed function pipeline as either the media or GPGPU pipeline, keeping in mind its general-purpose capability.

The GPU subsystem consists of an array of EUs and a set of shared functions outside the EUs. The EUs leverage these shared functions for I/O and dedicated computations. Programmers access the GPU subsystem via the 3D or media pipelines.

EUs support a rich instruction set. This instruction set has been optimized to support various 3D API shader languages, media functions processing, and compute kernels.

Shared functions are dedicated hardware accelerator units that provide specialized supplemental functionality for the EUs, to run filtered load, scatter, gather, and filter/blended store operations.

Programming the GPGPU Pipeline

The commands for either Programmable Media or GPGPU contexts uses mostly the same commands, with a few sequence differences as described in this section.

The Programmable Media Pipeline is programmed using command sequences. Media hardware threads are created through the parameterized media walker. Dispatch of a thread is controlled by a scoreboard mechanism.

The media pipeline can now use pipelined state changes, via the added MEDIA_STATE_FLUSH command. The MEDIA_STATE_FLUSH serves as a fence for state change by flushing the GPGPU Pipeline without waiting for threads to retire.

Note that media pipeline is the main user of the CURBE command. Media state commands deliver the interface descriptors directly, instead of loading them through indirect state.

Thread Lifecycle

Once a thread dispatcher (TS) selects a thread, the thread lifecycle is as follows:

- TS forwards the interface descriptor pointer to the L1 interface descriptor cache - a small fully associated cache containing up to four interface descriptors. If the interface descriptor is not found in the L1 interface descriptor cache, a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requested order.
- Once TS receives an interface descriptor, it checks whether the maximum concurrent thread number has been reached to determine whether to make a thread dispatch request or to stall the request until another thread retires. If the thread requests the use of scratch memory, TS also generates a pointer into the scratch space.
- TS then builds the transparent header and the R0 header, and TS makes a thread request to the thread dispatcher.
- TS keeps track of each dispatched thread, and monitors messages from each thread, including resource dereference and/or thread termination. When TS receives a thread termination message, it can recover the scratch space and thread slot allocated to the specified thread. The URB handle for a terminated thread may also be dereferenced for future reuse. Note that URB handle dereference may occur before a thread terminates. For more details, see the Media Message section.

The following flowchart outlines the thread lifecycle process.

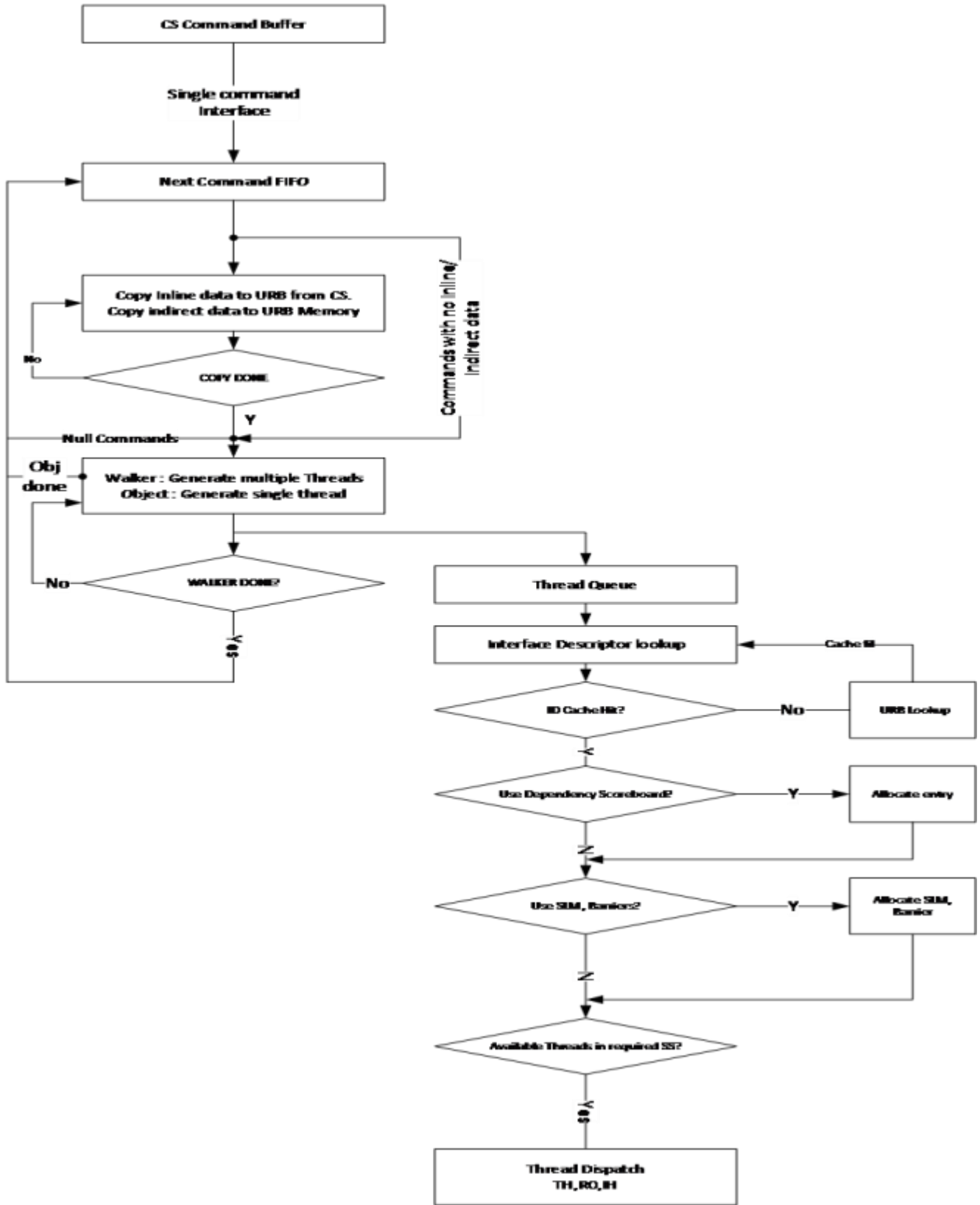


Figure: Thread Lifecycle Process



Command Sequence

The Command Sequence programming model for the GPGPU Pipeline (VFE/TSG) requires the following steps. Some of the steps listed are optional, but the order shown must be strictly followed. Some usage restrictions are emphasized for descriptive purposes. For more details, refer to the definitions of each command.

Step 1: MI_FLUSH/PIPE_CONTROL

- This step is mandatory.
- Each command in this step can be used multiple times, but that is not recommended for performance reasons.

Step 2: State command PIPELINE_SELECT

- This step is optional if it is known that, within the same context, the media pipeline was selected before Step 1.
- The command in this step can be used multiple time, but that is not recommended for performance reasons.

Step 3: State commands configuring pipeline states

STATE_BASE_ADDRESS:

- This command is mandatory for this step.
- Multiple instances of this command are allowed. The last instance overwrites previous ones.
- This command must precede any other state commands mentioned below.

The fields Indirect Object Base Address and Indirect Object Access Upper Bound are used to control indirect Media object load in VFE. The fields Dynamics Base Address and Dynamics Base Access Upper Bound are used to control indirect CURBE and interface descriptor object load in VFE.

On receiving the STATE_BASE_ADDRESS command, VFE stores the 48-bit Dynamics Base Address, Dynamics Base Access Upper Bound, Indirect Object Base Address, and Indirect Object Access Upper Bound in state registers. These values are used to calculate the fetch addresses for CURBE, Interface Descriptor, and Indirect Data respectively.

- Note that the STATE_BASE_ADDRESS command may be inserted before and after any commands listed in Steps 1 and 2 above. For example, STATE_BASE_ADDRESS may be placed in the ring buffer while other commands are put in a batch buffer.

STATE_SIP:

- This command is optional for this step, unless the kernels use SIP.

MEDIA_VFE_STATE:

- This command is mandatory for this step.
- This command destroys all outstanding URB handles in the system, then generates a new set of URB handles based on state parameters, number of URBs, and URB length from the VFE FF state.

- On receiving a MEDIA_VFE_STATE command, VFE allocates the number of handles specified by Number of URB Entries, with each handle size as specified in URB Entry Allocation Size, starting at CURBE Allocation Size + 64. These handles are stored in UEM.
- MEDIA_VFE_STATE also stores other state information, including Scratch Space, Gateway, Scoreboard, and Max Thread limit
- Multiple instances of the MEDIA_VFE_STATE command are allowed. The last instance overwrites previous ones.

MEDIA_CURBE_LOAD:

- This command is optional for this step.
- Multiple instances of this command are allowed. The last instance overwrites previous ones.
- On receiving a MEDIA_CURBE_LOAD command, VFE fetches the CURBE data from CURBE Data Start Address by reading CURBE Total Data Length bytes. The Address is defined relative to the Dynamics Base Address programmed through STATE_BASE_ADDRESS command.
- VFE breaks the CURBE Total Data Length into 64-byte chunks, and sends request through GAFS. The fetched data is written directly into URB at the address specified in the tag of the request. This address is set at 64. VFE tracks this fetch through its local up/down counter.

MEDIA_INTERFACE_DESCRIPTOR_LOAD:

- This command is mandatory for this step.
- Multiple instances of this command are allowed. The last instance overwrites previous ones.
- On receiving a MEDIA_INTERFACE_DESCRIPTOR_LOAD command, VFE fetches the Interface Descriptors from the Interface Descriptor Data Start Address by reading Interface Descriptor Total Length bytes. The Address is defined relative to the Dynamics Base Address, programmed through the STATE_BASE_ADDRESS command.
- VFE breaks the Interface Descriptor Total Length into 64 byte chunks and sends a request to GAFS. The fetched data is directly written into URB at the address specified in the tag of the request. That address is set as zero. VFE tracks this fetch through its local up/down counter.

Step 4: Primitive commands

This step must follow previous steps, which prepare the Media/GPU pipeline for the execution of primitive commands.

MEDIA_OBJECT/ MEDIA_OBJECT_GRPID:

- You can issue this command multiple times, to continue processing media primitives.

On receiving a MEDIA_OBJECT/MEDIA_OBJECT_GRPID command, VFE fetches indirect data from Indirect Data Start Address by reading Indirect Data Length bytes. The Address is defined relative to the Indirect Object Base Address, programmed through the STATE_BASE_ADDRESS command.

VFE breaks the Indirect Data Length into 64-byte chunks and sends the request to GAFS. The fetched data is written directly into URB at the address specified in the request tag. This address is set as the

handle address + inline data length (found from `csdwlengthr [] - cmd_length`), rounded up to the next 32-byte boundary. VFE tracks this fetch through its local up/down counter.

- If inline or indirect data is present in the command, the URB handle is allocated from UEM.
- Upon receiving this command, VFE stores the Media Object information to pass to the TSG unit, including the Interface Descriptor Offset, Child Present, Thread Synchronization, Scoreboard, and GRPID (for `MEDIA_OBJECT_GRPID` command).
- The command writes inline data at the starting URB handle address through the GAFS interface. If the inline data is not a multiple of 32-bytes, padding is added to the next 32-byte boundary.
- When the transfer of inline and indirect data is complete, VFE sends a thread dispatch signal to TSG.

GPGPU Pipeline Notes

- Set up the interface descriptor with the number of threads in barrier. The barrier ID is not specified here, because hardware can automatically assign barriers to available thread groups. The amount of CURBE data to deliver per thread dispatch is set in the interface descriptor.
- Set up CURBE with thread IDs and common data for all thread dispatches in the thread group.
- Set up a GPGPU_WALKER command with the thread group IDs to dispatch the threads. Each thread dispatch in the thread group sends CURBE data in sections; a new thread group starts sending the CURBE data from the beginning of the buffer.
- Place a `MEDIA_STATE_FLUSH` right before the `MI_BATCH_BUFFER_END` of any batch buffer that uses `MEDIA_OBJECT`.
- The kernel should handle barriers as follows:
 - The `BarrierMsg` contains the barrier ID and a way to reprogram the barrier count.
 - The barrier count reprogram is not normally used for GPGPU workloads.
 - When all threads in the group have reached the barrier, the gateway returns a notification bit 0.
 - The kernel must wait for the barrier to finish with a `WAIT NO`.

CURBE/Indirect Payload Dispatch

Example of a CURBE command sequence:

- `MEDIA_STATE_FLUSH` // Make sure dispatch is not accessing CURBE
- `MEDIA_CURBE_LOAD`
- `GPGPU_WALKER`

The GPGPU commands extend to allow indirect input as an alternative to CURBE using the exact same mechanism: the same offset specifies what data is delivered to all threads, and a count specifies how much data is delivered per thread. A single indirect pointer points to both the Cross-Thread and Per-Thread Constant Data, which are stored in the URB. The positions of the cross-thread and per-thread constants are swapped in the EU GRF.

To use indirect payload storage, the product (URB entry allocation size * number of URB entries) must be enough to cover the sum of the cross-thread and per-thread indirect data to be loaded. The MEDIA_VFE_STATE command sets these state variables. The URB entry allocation must be equal to or greater than the indirect data length in GPGPU_WALKER. Use the maximum number of URB entries that fills the URB space, to avoid a shortage of handles that could cause a performance problem.

Indirect input is defined as follows:

- Indirect uses a 32-bit memory pointer for the start address (compared to 64 bit for CURBE).
- Indirect does not use the constant URB entry read offset.
- Indirect multiplies the cross-thread constant data read length and constant entry read length by 32 to convert them into bytes.
- The X/Y/Z payload in the EU GRF comes before the cross-thread constant data.

Media GPGPU Payload Limitations

The Media/GPGPU instructions can have three types of payloads, but not all combinations are allowed. The following table lists the legal combinations.

Table: Media GPGPU Payload Limitations

Workload	Commands	Data Stored
GPGPU	GPGPU_WALKER	CURBE
	GPGPU_WALKER	INDIRECT
Media (Legacy)	Media_Object	CURBE
	Media_Object	INDIRECT
	Media_Object	INLINE
	Media_Object	CURBE+INLINE
	Media_Object	CURBE+INDIRECT
	Media_Object	INLINE+INDIRECT
	Media_Object	CURBE+ INLINE+INDIRECT
	Media_Object_Walker	CURBE
	Media_Object_Walker	INLINE

	Media_Object_Walker	CURBE+INLINE
	Media_Object_Walker	INDIRECT
Media using Barrier/SLM	Media_Object_GRPID	CURBE
	Media_Object_GRPID	INDIRECT
	Media_Object_GRPID	INLINE
	Media_Object_Walker (with group id)	CURBE
	Media_Object_Walker (with group id)	INLINE
	Media_Object_Walker (with group id)	INDIRECT

The Dynamic State MOCS, specified in **STATE_BASE_ADDRESS**, fetch indirect and CURBE payloads from memory during thread dispatch.

Media State Model

The media state model is based on an in-line state load mechanism. State commands load VFE state, URB configuration, and Interface Descriptors to VFE hardware.

All Interface Descriptors have the same size and are organized as a contiguous array in memory. The Interface Descriptor Index can select these descriptors for a given kernel, which allows different kinds of kernels to coexist in the system.

The state-command-related headers are as follows.

Pipeline (Media) Bits[28:27]	Opcode Bits[26:24]	Sub Opcode Bits[23:16]	Command
2h	0h	00h	MEDIA_VFE_STATE
2h	0h	01h	MEDIA_CURBE_LOAD
2h	0h	02h	MEDIA_INTERFACE_DESCRIPTOR_LOAD

Commands for GPGPU Pipe

This section lists and explains the various commands that VFE executes for media and GPGPU context on render pipe, including several commands that are shared with media mode. There are three types of VFE commands:

- Null command - may write URB, but does not place any request to the TSG unit.
- Object command - may write URB, and then place a single request to the TSG unit.
- Walker command - may write URB, and then trigger VFE_WALKER to generate multiple requests to the TSG unit.

Table: VFE Commands

CmdID	Command	Command type
1	MEDIA_CURBE_LOAD	null command
2	MEDIA_INTERFACE_DESCRIPTOR_LOAD	null command
3	MEDIA_VFE_STATE	null command
3	MEDIA_STATE_FLUSH ¹	object command
4	MEDIA_OBJECT	object command
5	MEDIA_OBJECT_GRPID	object command
7	MEDIA_WALKER	walker command
9	GPGPU_WALKER	walker command

¹ The MEDIA_STATE_FLUSH command is also an object command, to ensure that it reaches the top of the root FIFO in TSG unit only when all previous requests are dispatched.



MEDIA_VFE_STATE

MEDIA_VFE_STATE is the primary state command for GPGPU pipe, and sets up context for GPGPU pipe usage as follows:

VFE allocates URB handles (1-128) and specifies the size for each URB entry.

Defines Scratch Space allocation per thread and CURBE allocation size.

Sets up and controls the hardware scoreboard in the media pipeline. Sets up the type, mask and relative vertical and horizontal distance of the dependent instant for each of the eight possible scoreboards.

Sets up the gateway usage model, especially if the gateway protocol needs to be bypassed and timestamp reset.

Defines the maximum number of simultaneous active threads possible in the system and in the thread dispatch policy, chooses the best subslice for a thread, and selects a field to specify which slices or subslices the dispatch can disable.

Defines SLM bank selection policy, determined by a bit that allows selection between the least loaded SLM bank or a bank that is paired with the thread's subslice.

Note that a stalled PIPE_CONTROL (a top-end synchronization command) is required before the MEDIA_VFE_STATE command, unless the only bits changed are scoreboard-related. If bit changes are scoreboard-related, a MEDIA_STATE_FLUSH command is sufficient.

For more details, see the MEDIA_VFE_STATE definition.

MEDIA_STATE_FLUSH

The MEDIA_STATE_FLUSH command flushes the media/GPGPU parser. In general this command is the same as an MI_FLUSH, except MI_FLUSH clears the entire 3D/render pipeline.

A key function of the MEDIA_STATE_FLUSH command is to stall the Command Streamer (CS) in one of two ways based on a watermark:

- (1) Stalls the CS until there is programmed amount of room in a subslice, or enough SLM, or a free barrier.
- (2) Stalls the CS until the last-level thread dispatch unit is able to dispatch the last thread to the EU.

The MEDIA_STATE_FLUSH command also indexes the resource needed to support the above watermark. A bit in this command indicates whether the write data out of the current thread group needs to be flushed beyond the visibility of following commands.

Note: You should program the MEDIA_STATE_FLUSH command for any of the following:

- (a) prior to new media state, CURBE, or interface descriptor commands;
- (b) while programming for a new state in the same context; or
- (c) while switching to a new context.

Also note that the interface descriptor offset field updates the MEDIA_STATE_FLUSH command to specify all resources required for the next thread group. If the resources are not available, the next thread group cannot start.

For more details, see the MEDIA_STATE_FLUSH definition.

MEDIA_VFE_STATE

MEDIA_VFE_STATE is the primary state command for GPGPU pipe, and sets up context for GPGPU pipe usage as follows:

- VFE allocates URB handles (1-128) and specifies the size for each URB entry.
- Defines Scratch Space allocation per thread and CURBE allocation size.
- Sets up and controls the hardware scoreboard in the media pipeline. Sets up the type, mask and relative vertical and horizontal distance of the dependent instant for each of the eight possible scoreboards.
- Sets up the gateway usage model, especially if the gateway protocol needs to be bypassed and timestamp reset.
- Defines the maximum number of simultaneous active threads possible in the system and in the thread dispatch policy, chooses the best subslice for a thread, and selects a field to specify which slices or subslices the dispatch can disable.
- Defines SLM bank selection policy, determined by a bit that allows selection between the least loaded SLM bank or a bank that is paired with the thread's subslice.



Note that a stalled PIPE_CONTROL (a top-end synchronization command) is required before the MEDIA_VFE_STATE command, unless the only bits changed are scoreboard-related. If bit changes are scoreboard-related, a MEDIA_VFE_STATE command is sufficient.

For more details, see the **MEDIA_VFE_STATE** definition.

MEDIA_STATE_FLUSH

The MEDIA_STATE_FLUSH command flushes the media/GPGPU parser. In general, this command is the same as an MI_FLUSH, except MI_FLUSH clears the entire 3D/render pipeline.

A key function of the MEDIA_STATE_FLUSH command is to stall the Command Streamer (CS) in one of two ways based on a watermark:

- (1) Stalls the CS until there is programmed amount of room in a subslice, or enough SLM, or a free barrier.
- (2) Stalls the CS until the last-level thread dispatch unit is able to dispatch the last thread to the EU.

The MEDIA_STATE_FLUSH command also indexes the resource needed to support the above watermark. A bit in this command indicates whether the write data out of the current thread group needs to be flushed beyond the visibility of following commands.

Note: You should program the MEDIA_STATE_FLUSH command for any of the following:

- (a) prior to new media state, CURBE, or interface descriptor commands;
- (b) while programming for a new state in the same context; or
- (c) while switching to a new context.

Also note that the interface descriptor offset field updates the MEDIA_STATE_FLUSH command to specify all resources required for the next thread group. If the resources are not available, the next thread group cannot start.

For more details, see the **MEDIA_STATE_FLUSH** definition.

MEDIA_CURBE_LOAD

The MEDIA_CURBE_LOAD command establishes the Constant URB Entry (CURBE) mode for Payloads by defining:

Total CURBE data length provides an address range up to 2^{17} bytes for the CURBE payload data. This field is 64-byte aligned.

CURBE start address specifies the 64-byte aligned address where CURBE data must start to reside. This is a relative address to the address established by the STATE_BASE_ADDRESS command.

For more details, see the **MEDIA_CURBE_LOAD** definition.

MEDIA_INTERFACE_DESCRIPTOR_LOAD

The MEDIA_INTERFACE_DESCRIPTOR_LOAD command sets up the VFE by defining where to fetch the Interface descriptor and how many Bytes to fetch:

Loads the descriptor total length in bytes and the 64-byte aligned pointer to the interface descriptor data. This interface descriptor pointer is relative to the Dynamic Base Address programmed through the STATE_BASE_ADDRESS command.

VFE fetches the data based on the pointer and loads the data into URB.

Note: Use a MEDIA_STATE_FLUSH (without watermark) before this command to ensure that the temporary interface descriptor storage is cleared.

For more details, see the **MEDIA_INTERFACE_DESCRIPTOR_LOAD** definition.

MEDIA_OBJECT

The MEDIA_OBJECT command is the basic media primitive command for the media pipeline, defined as follows:

Supports loading of inline, CURBE, and indirect data.

Can be used in the following three VFE modes: Generic mode, IS mode, and VLD mode.

Specifies the length of inline data in all of the above three modes and all the associated rules.

For indirect data, defines the rules for determining the length and start address for the same three modes.

Cannot be used in AVC-IT, AVC-MC and VC1-IT modes.

Note: If neither SLM nor a barrier is used, MEDIA_OBJECT must be used instead of the MEDIA_OBJECT_GRPID command.

For more details, see the **MEDIA_OBJECT** definition.

MEDIA_OBJECT_GRPID

The MEDIA_OBJECT_GRPID command is a variation of the MEDIA_OBJECT command and includes a 32-bit group ID, which is a unique identifying number describing threads that share the same barriers and/or SLMs.

- The interface descriptor command specifies how much SLM is needed and how many threads report to a barrier. This command tracks and executes such groups.
- All MEDIA_OBJECT_GRPIDs with the same group ID need to have the same interface descriptor. They must also be dispatched to the same Subslice, which is ensured by the hardware dispatcher if Force Destination = 0, or by software if Force Destination = 1.

- Slice/SS destination select parameters can ensure where the dispatch happens for the group. Software can also ensure that all threads needed for the Barrier fit into a subslice, or the Barrier would never be satisfied due to insufficient condition.
- Hardware only dispatches a single group at a time, and that must be enforced by the programmer for MEDIA_OBJECT_GRPID commands - unlike the MEDIA_OBJECT_WALKER command.
- A bitfield defines whether threads associated with MEDIA_OBJECT_GRPID need to use the scoreboard, and whether there are supporting fields to provide the X, Y, and color terms for the scoreboard.

Note: MEDIA_OBJECT_GRPID supports the GPGPU version of payload delivery using the following:

- Per-thread payload containing a split of either indirect or CURBE between the threads, and
- Cross-thread payload sent to all threads.

For indirect payload splitting to work properly, you must send the same pointer with all the commands associated with the thread group. A kernel does not split inline data, but sends the payload attached to each command with the corresponding thread. Only one of inline, indirect, or CURBE is allowed, and you must send at least one form of payload.

Also note that you must place MEDIA_STATE_FLUSH, with the watermark bit, between groups created by MEDIA_OBJECT_GRPID. The Interface Descriptor associated with the watermark must match the Interface Descriptor used for the subsequent group.

There are two walker commands: MEDIA_OBJECT_WALKER and GPGPU_WALKER.

For more details, see the **MEDIA_OBJECT_GRPID** definition.

MEDIA_OBJECT_WALKER

The MEDIA_OBJECT_WALKER command can create multiple threads with various patterns, per the walker parameters provided in the command. Thread generation for "parameterized media walker" works as follows:

The MEDIA_OBJECT_WALKER command uses the VFE hardware walker to generate threads associated with a rectangular-shaped object. You can use this command only in generic mode. This command can generate one or more threads.

MEDIA_OBJECT_WALKER supports loading only of inline data or CURBE, but not indirect data. You must send at least one form of payload.

When inline data is present, MEDIA_OBJECT_WALKER repeats that data for all dispatched threads, so pipeline flushes are not required. When commands are issued without a pipeline flush in between, changes to the 'global' (shared) data are supported. (Unlike CURBE, which requires pipeline flushes for such changes.)

Note: You must place a MEDIA_STATE_FLUSH command after each MEDIA_OBJECT_WALKER with inline data.

For more details, see the **MEDIA_OBJECT_WALKER** definition.

MEDIA_WALKER

Each MEDIA_WALKER command dispatches multiple threads covering the (X,Y) dimensions of an image frame.

MEDIA_WALKER is similar to MEDIA_OBJECT_WALKER command, but is enhanced so that multiple MEDIA_WALKER commands can be executing at the same time from the same command stream. The number of simultaneously active MEDIA_WALKER commands is specified in MEDIA_VFE_STATE command. Simultaneously active MEDIA_WALKER commands dispatch their threads to non-overlapping DSS.

Like MEDIA_OBJECT_WALKER, the dispatch order of the MEDIA_WALKER threads is programmable for various patterns using inner, outer, local, and global loop parameters. Furthermore, a color parameter can be specified to support the simultaneous processing of 2 or more frames using the same parameters. This is useful because the inter-thread dependencies can significantly reduce the number of threads actively running in the walker command.

MEDIA_WALKER specifies the Interface Descriptor as the first part of its Indirect Data. Each dispatched thread receives a payload that is specified in the remainder of the Indirect Data.

For more details, see the definition.

GPGPU Walker

A single GPGPU job may require thousands or even millions of threads. Creating these threads individually would be inefficient, so the GPGPU_WALKER command generates the threads algorithmically (rather than modifying the Media Walker).

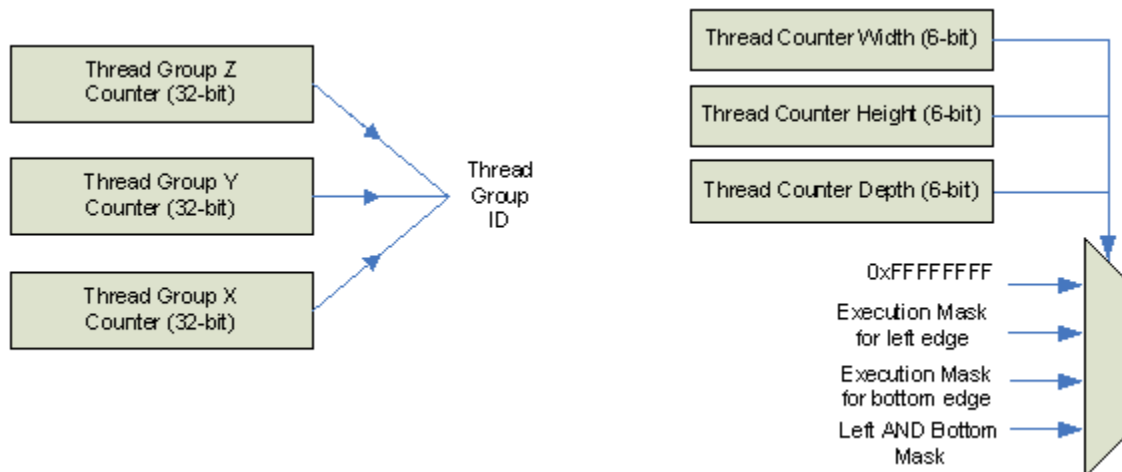


Figure: GPGPU Thread Group Walker

The X, Y, and Z counters for a thread group each have an initial and maximum value. Combining the three counter values forms the thread group ID sent with each dispatch. Each counter contains 32 bits; there is no limit on the size of the thread ID.

The GPGPU requires that 1024 SIMD channels be available for a maximum-size thread group. Thread groups that do not need access to a barrier or SLM are not limited to 1024 SIMD channels.

In general, the formula for "number of Threads" = "number of Channels" / SIMD size.

For example, in a system with ten EUs per subslice, each with seven hardware threads, a maximum-size thread group fits in a subslice if SIMD16 instructions are used, but not if SIMD8 instructions are used.

See the Configurations section for an exact determination of what SIMD option is required to fit in the subslice of the targeted configuration.

The three thread counters count the number of dispatches in a single thread group: up to 32 dispatches for SIMD32 or 64 dispatches for SIMD16/8. There are three thread counters to select the execution masks correctly - see the Execution Masks section.

Each thread counter is six bits to allow flexibility for any dimension going to 64, while the rest do not increment. A thread is generated each time one of the thread counters increment. When the thread group X counter reaches the maximum it is reset to 0, and the Y counter is incremented. When the thread group Y counter reaches the maximum it is reset to 0, and the Z counter is incremented. When all the counters reach their maximum values, the thread group is done and the thread group counter can increment to start a new thread group.

The compiler determines how many SIMD channels are needed per thread group, and then decides how these are split among EU threads. The number of threads is programmed in the thread counter, and the SIMD mode (SIMD8/SIMD16/SIMD32) is specified in the GPGPU_WALKER command.

The maximum thread group size must fit into a single subslice (or DSS) and run in parallel, so the number of EU threads must be less than the number specified in Configurations for threads per subslice (DSS).

Thread Payload

The payload for each thread dispatched is as follows:

1. A thread group ID that identifies the group to which the threads belong. This consists of three 32-bit X/Y/Z values.
2. The set of X/Y/Z counters that form the thread ID for each channel. If the Z counter is not used, then only X and Y are needed.
3. The execution mask that indicates what channels are active.

Thread IDs form a 2D or 3D surface that is mapped into SIMD32, SIMD16, or SIMD8 dispatches. The compiler supplies the mapping, rather than having the hardware force a particular mapping of thread IDs to channels.

The VFE receives a simple count of the number of threads per thread group, which counts the number of dispatches. A constant buffer in the MEDIA_CURBE_LOAD command contains the thread IDs for all threads in a thread group. You can use a single set of thread IDs repeatedly for all thread groups, since the thread IDs are the same for each thread group ID output by the GPGPU_WALKER.

The following figure is an example set of payloads for a 2 Z x 2 Y x 12 X and a SIMD16 dispatch. This thread group requires three dispatches:

3 2 1 0 11 10 9 8 7 6 5 4 3 2 1 0	Thread id X for dispatch 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0	Thread id Y for dispatch 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Thread id Z for dispatch 0
7 6 5 4 3 2 1 0 11 10 9 8 7 6 5 4	Thread id X for dispatch 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1	Thread id Y for dispatch 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0	Thread id Z for dispatch 1
11 10 9 8 7 6 5 4 3 2 1 0 11 10 9 8	Thread id X for dispatch 2
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0	Thread id Y for dispatch 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	Thread id Z for dispatch 2

Figure: Dispatch Arrangement

In this case the thread counter width is programmed with a maximum value of three. Since the execution masks are all F, it doesn't matter how the thread counters are programmed, as long as they count to three before finishing the thread group.

The first dispatch would tell the TS that the payload starts at the beginning of the constant buffer and has a length of three. The next dispatch would have a payload starting at Constant_Buffer_Start + 3. The final dispatch payload starts at Constant_Buffer_Start + 6. If there are more thread groups in the command they would each get exactly the same payload, except for a different thread group ID, barrier, and shared local memory space.

Execution Masks

The number of channels required by a GPGPU job may not fit evenly into the number of SIMD channels, which can leave some channels idle. The execution mask tells the hardware what channels are actively used.

The payload for a general-purpose thread must include the execution mask. The execution mask has one bit set per channel for up to 32 channels. SIMD16 and SIMD8 use the LSB bits of the execution mask. The 5-bit number transferred from VFE expands to produce the 32-bit mask. This mask uses the Dmask from the pixel shader dispatch in the transparent header.

A thread group is modeled as a 3D solid, with each channel acting as one X/Y/Z point in the solid. This group can have the following forms:

A line with 1024 X channels from 0 to 1023, and constant Y/Z;

A square with X ranging from 0 to 32 and Y ranging from 0 to 32;

A cube with X ranging from 0 to 9, Y ranging from 0 to 9, and Z ranging from 0 to 9.

Software needs to determine how to map these shapes onto the 32 SIMD32 channels per dispatch, or 16 SIMD16, or 8 SIMD8. The mapping per thread is a 2D square of channels such as 8x4, 16x2, or 32x1.

Below is a diagram of a 22x6 thread group that is mapped onto a set of 8x4 SIMD32 channels:

8x4	8x4	8x4	
8x4	8x4	8x4	
8x4	8x4	8x4	

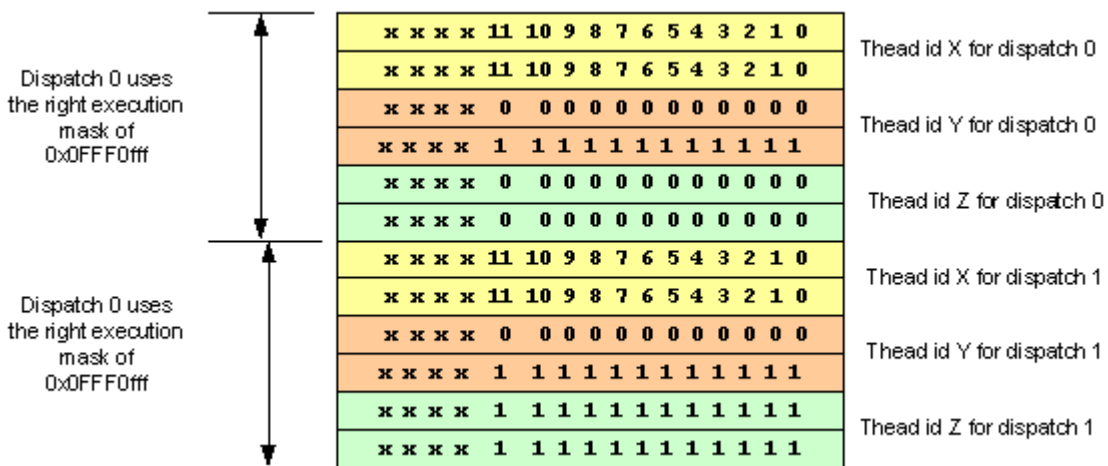
Figure: Mapping onto a Set of SIMD32 Channels

Note that the dispatches in the upper left have execution masks of all-F, while all the right edge dispatches have the same execution mask; likewise, all the bottom edge dispatches have the same execution mask. The bottom right is the logical-AND of the right and bottom edge dispatches.

A 32-bit right and bottom mask is sent with the GPGPU_WALKER command, and the thread width, height and depth counters determine when the masks are used. Width, height and depth are used instead of X/Y/Z, since it is not required that width = X; width and height are the two variables that are changing in a single SIMD dispatch even if they are Y and Z.

For each dispatch, the width counter increments until it reaches the maximum. The dispatch with width=max uses the right execution mask. The height counter is then incremented, and the process repeated. If at any time the height counter = max then the execution mask is the bottom execution mask. When the height and width counters are both max then the dispatch is the AND of the right and bottom masks, and the depth counter increments.

The same 2Z x 2Y x 12X thread group described above dispatched as SIMD32 with each dispatch delivering a 16X x 2Y shape requires two dispatches with empty bits in the right execution mask and all F in the bottom.



The width and height counter would have a maximum of one, and the depth counter would have a maximum of two. The two dispatches use the AND of the two masks, but since the bottom mask is F it would be the same as just the right mask.

The execution masks can also be used when the software wants to pack the channels rather than lay them out in a regular pattern:

3 2 1 0 11 10 9 8 7 6 5 4 3 2 1 0	Thread id X for dispatch 0
7 6 5 4 3 2 1 0 11 10 9 8 7 6 5 4	
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0	Thread id Y for dispatch 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Thread id Z for dispatch 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0	
11 10 9 8 7 6 5 4 3 2 1 0 11 10 9 8	Thread id X for dispatch 1
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0	Thread id Y for dispatch 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	Thread id Z for dispatch 1

In this case the width counter can have a maximum of two, and the height and depth counters have a maximum of one. The first dispatch uses the bottom mask only (all F), and the second uses the right AND bottom mask to remove unused channels.

Synchronization of the Media-GPGPU Pipeline

The PIPE_CONTROL command synchronizes the Media/GPGPU Pipeline in the same way as for the 3D pipeline. PIPE_CONTROL for the Media pipe case stalls 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 then occur per the enabled PIPE_CONTROL parameter fields. Due to the stalling behavior at the top, only one PIPE_CONTROL command can be outstanding at a time on the Media Pipe.

The top of the pipe synchronization enforces the read-only cache invalidation. This synchronization guarantees that primitives rendered after a synchronization event fetch the latest read-only data from memory.

For more details, see the **PIPE_CONTROL** command definition.

Supporting Commands for MEDIA-GPGPU Pipe

PIPE_CONTROL

Use the **PIPE_CONTROL** command to implement synchronization in the render pipe. For Media and GPIPE, this causes a synchronization that stalls at the top of the pipe until the Media FFs finish processing commands parsed before PIPE_CONTROL.

PIPE_CONTROL with flush replaces MI_FLUSH.

PIPELINE_SELECT

Use the **PIPELINE_SELECT** command to select the currently active pipe in the render engine. Only one of either the 3D pipeline or the GPGPU pipeline can be selected at a time; selection of these pipelines is mutually exclusive.



STATE_BASE_ADDRESS

The **STATE_BASE_ADDRESS** command sets the base pointers for subsequent state, instruction, and media indirect object accesses by the GPPIPE.

3DSTATE_BINDING_TABLE_POOL_ALLOC

The **3DSTATE_BINDING_TABLE_POOL_ALLOC** command when enables Binding Table Pool sets the base pointer for subsequent Binding Table Pointer access by the GPPIPE.

Thread Spawner (TS)

Thread spawning is fundamental to the functionality of GPGPU and programmable media. The VFE can create sets of threads using the GPGPU_WALKER, or the VFE can assemble general-purpose threads using MEDIA_OBJECT_WALKER in generic media mode.

A command streamer interprets and forwards VFE-decodable commands to the VFE unit. The VFE first runs one set of commands to set up the GP pipe, then runs WALKER commands to do automatic thread generation. A single GPGPU WALKER command can create a whole list of threads by walking either a one-dimensional range (X-pattern), a two-dimensional range (X-Y pattern), or a three-dimensional range (X-Y-Z pattern). Similarly, a media WALKER command can create a list of threads through a walking pattern of left, right, diagonal, forward, or backward.

The thread request queue stores thread requests and is in arbitration only if the number of outstanding threads does not exceed the maximum thread state variable. Otherwise, the thread request queue stalls until some other threads retire or terminate. TS keeps everything needed to get the threads ready for dispatch, and then tracks dispatched threads until their retirement.

Memory Resources for Threads

The GPGPU memory model uses URB, SLM, scratch space, and GRF. Hardware, URB, SLM, and scratch space are physically in the L3 Cache and logically segregated; GRF is exclusively inside the EUs. The following section covers these items and their usage model in more detail.

URB

The Unified Return Buffer (URB) is as a temporary holding area that handles the process of thread creation and dispatch. VFE manages the URB in both GPGPU and generic media modes and allocates one URB handle for each thread. The URB handle delivers the payload into the thread. The TS signals VFE to de-reference the URB handle immediately after receiving acknowledgement from TD that the thread has been dispatched.

The first 64 URB entries are reserved for the interface description, and CURBE data is placed after the IDs. URB handles are required for indirect data. When the VFE starts, it creates up to 128 handles by partitioning the remaining URB space into evenly spaced addresses, and saving the resulting handles to a buffer. The resulting handles are removed from the buffer on use and returned to the buffer when they retire.

MEDIA_VFE_STATE specifies the amount of CURBE space, the URB handle size, and the number of URB handles.

The driver must ensure that:

$$((\text{URB_handle_size} * \text{URB_num_handle}) - \text{CURBE} - 64) \leq \text{URB_allocation_in_L3}.$$

Shared Local Memory (SLM)

The Shared Local Memory is a 64KB block per subslice that must be shared between all thread groups on that half-slice. An SLM memory manager, similar to the scratch space memory manager, allocates the SLM space.

At a given time, a single interface descriptor dispatches most threads. For a new interface descriptor request, the GP pipe is drained, and all shared memory recovered before allocating new shared memory. Hence only a single size of shared memory needs to be supported at one time.

For simplicity, only power-of-2 shared memory sizes from 4KB to 64KB are allowed. A thread request specifies what size is needed. The first thread of a Thread Group is marked whether it requires new shared local memory. If not, the old SLM offset is sent with the dispatch.

A simple set of 16 bits allocates 4KB shared memory, with more bits used for larger sizes. A priority encoder finds the first unused bit and the offset associated with a particular barrier ID. The barrier ID then tracks the thread group. When thread group tracking indicates that a thread group is completely retired, that section of SLM can be reclaimed.

Local Memory/Scratch Space

Local memory is allocated per thread dispatched to the EU. The scratch space manager provides between 1KB and 2MB memory per thread.

GRF

A General Register File (GRF) inside the EU represents the per-thread private memory of the general-purpose compute framework memory model. The GRFs are register files dedicated to read/write operations of the threads. There are 128 GRFs of 32 bytes each per thread.

Media and GPGPU Indirect Thread Dispatch

Indirect thread dispatch allows one thread group to control the group size of a following thread group. To enable GPGPU indirect thread dispatch, use the following sequence of commands in the ring buffer:

GPGPU_OBJECT/WALKER	// Either a set of objects or a walker to dispatch a thread group to write the next group's properties to memory.
MI_FLUSH	// Make sure the thread group has finished executing.
MEDIA_CURBE_LOAD	// Load the thread ids for new group.

MI_LOAD_REGISTER_MEMORY	// Load the indirect MMIO GPGPU registers from the mem written by the previous group.
GPGPU_WALKER (indirect)	// A walker with the indirect bit set.

The first thread group writes the following data to memory:

1. The thread IDs delivered in the CURBE, written where the following MEDIA_CURBE_LOAD reads them.
2. The GPGPU_WALKER parameters, written where the MI_LOAD_REGISTER_MEMORY reads them:
 - a. GPGPU_DISPATCHDIMX - the X dimension of the number of thread groups in which to dispatch; DW offset 7
 - b. GPGPU_DISPATCHDIMY - the Y dimension of the number of thread groups in which to dispatch; DW offset 10
 - c. GPGPU_DISPATCHDIMZ - the Z dimension of the number of thread groups in which to dispatch; DW offset 12

Use CURBE (rather than indirect) for the payload when using indirect dispatch.

Parameterized Media Walker

The Parameterized Media Walker is a hardware thread-generation mechanism to create threads associated with units in a generalized two-dimensional space (e.g. blocks in a 2D image). With a small number of unit step vectors, the walker can implement a large number of walking patterns. This walker may provide functions that are normally handled by the host software, thus simplifying the host software to GPU interface.

The walker described here is doubly nested, meaning a "local" walker can perform a variety of two-dimensional walking patterns while a "global" walker performs similar two-dimensional patterns upon many local walkers. The local walker has three levels (outer, middle, and inner) while the global walker has two levels (outer and inner). Hence the walker algorithm has five-nested loops that modify local state based on user-defined unit step vectors.

The parameterized media walker's programmability is derived from the following.

- The walker traverses a unit-normalized surface. Some example unit sizes:
 - 1x1: Walking pixels,
 - 4x4: Walking sub-blocks,
 - 16x16: Walking macro-blocks, and
 - 32x16: Walking macro-block-pairs.
- Unit step vectors to describe the motion at each of level of nesting.
- Starting locations for the local and global walkers.
- Block sizes of the local and global walker.

- A small number of special mode controls for the inner-most loop, designed to efficiently divide an image into two balanced workloads for dual-slice designs.

Walker Parameter Description

The global and local loops are both described by the same four parameters:

- Resolution,
- Starting location,
- Outer unit vector, and
- Inner unit vector.

The local inner loop has some special modes that are described later. A table of the user inputs and some example values are shown below:

GLOBAL LOOP PARAMETERS							
Global Resolution		Global Start		Outer Loop Unit Vector		Inner Loop Unit Vector	
X	Y	X	Y	X	Y	X	Y
120	68	0	0	32	0	0	32
LOCAL LOOP PARAMETERS							
Block Resolution		Local Start		Outer Loop Unit Vector		Inner Loop Unit Vector	
X	Y	X	Y	X	Y	X	Y
32	32	0	0	1	0	-2	2
LOCAL INNER LOOP SPECIAL MODE SELECTS							
Dual Mode	Repel	Attract			ExtraSteps	X	Y
TRUE	FALSE	FALSE			1	0	1

A programmer implicitly defines what a "unit" represents, which means the walker traverses a "unit normalized space" that is not inherently bound to pixel walking.

If the smallest unit of work to be walked is a 4x3 block of pixels, you can program the inner loop to step (4,3) or (1,1) as follows:

- In the (4,3) case, the user is walking units of pixels.
- In the (1,1) case, the user is walking units of 4x3 blocks of pixels.

Note that hardware doesn't contain enough bits for pixel walking for resolutions like 1920x1088 pixels. The intended usage of the walker is for block walking where the block size is not relevant to the walker parameters.

Basic Parameters for the Local Loop

Basic Parameters for the Local Loop The local inner and outer loop xy-pair parameters alone can describe a large variety of primitive walking patterns. Below are nine primitive walking patterns generated by varying only the inner and outer unit step vectors of the local loop:

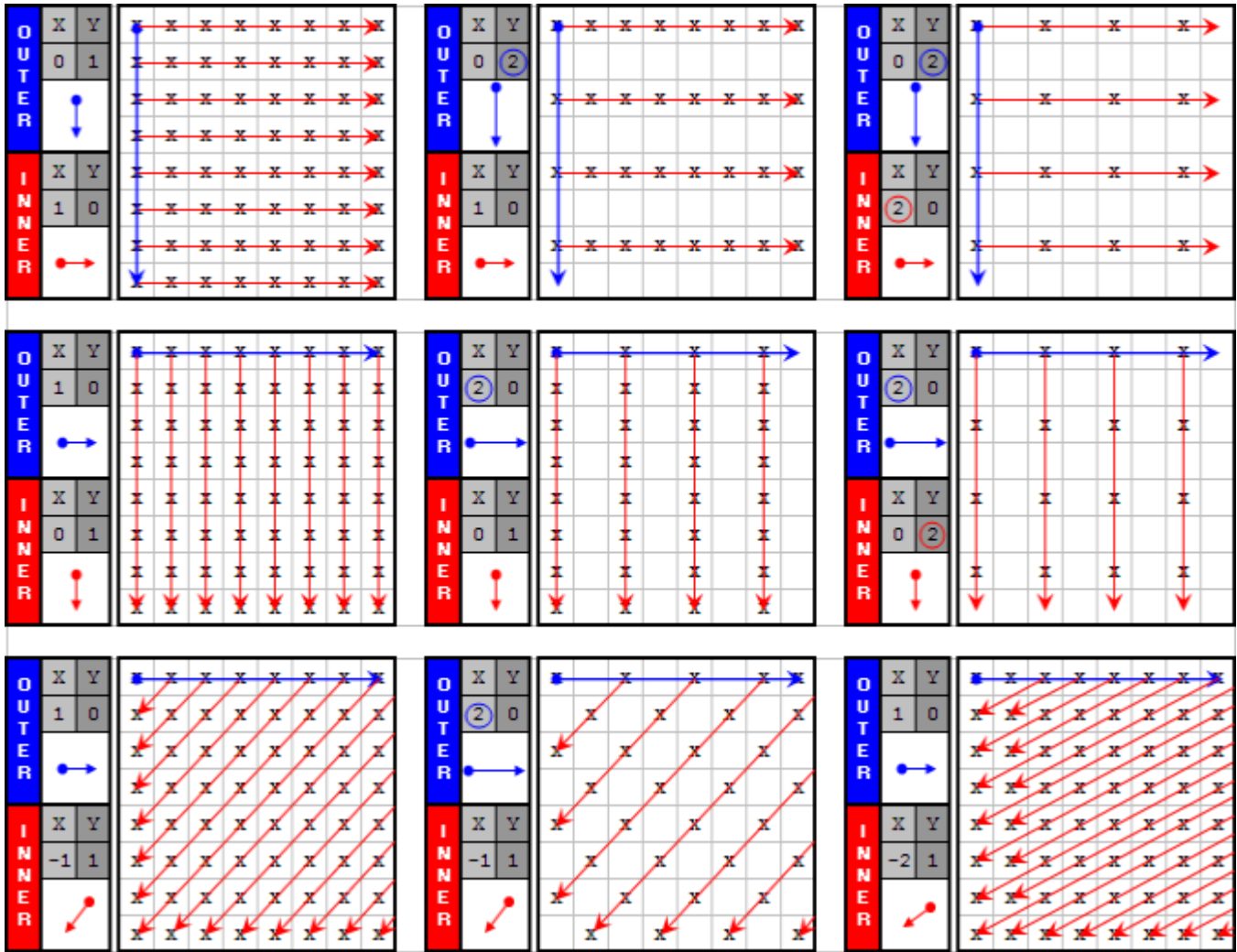


Figure: Local Loop Walking Patterns

- The top row of the figure above shows the outer unit vector pointing down (+Y) and the inner unit vector pointing right (+X). Skip rows and columns by increasing the unit step vectors beyond one.
- The middle row of the figure shows the outer unit vector pointing right (+X) and the inner unit vector pointing down (+Y). Skip rows and columns by increasing the unit step vectors beyond one.
- The last row of the figure shows the option to walk patterns not aligned to the edges. The left option yields a 45° walking pattern by setting the inner unit vector to (-1,1). The middle option yields a checkerboard pattern by skipping every other outer loop and retaining the inner unit vector of (-1,1). The right option yields a 26.5° walking pattern by setting the inner unit vector to (-2,1).

The block resolution, shown as [8,8], and the starting location, shown as [0,0], can be varied and the walking patterns can be stretched and rotated in many ways. The diagram below shows an example of where the start position and unit step vectors can be set to achieve a full rotation of the same pattern:

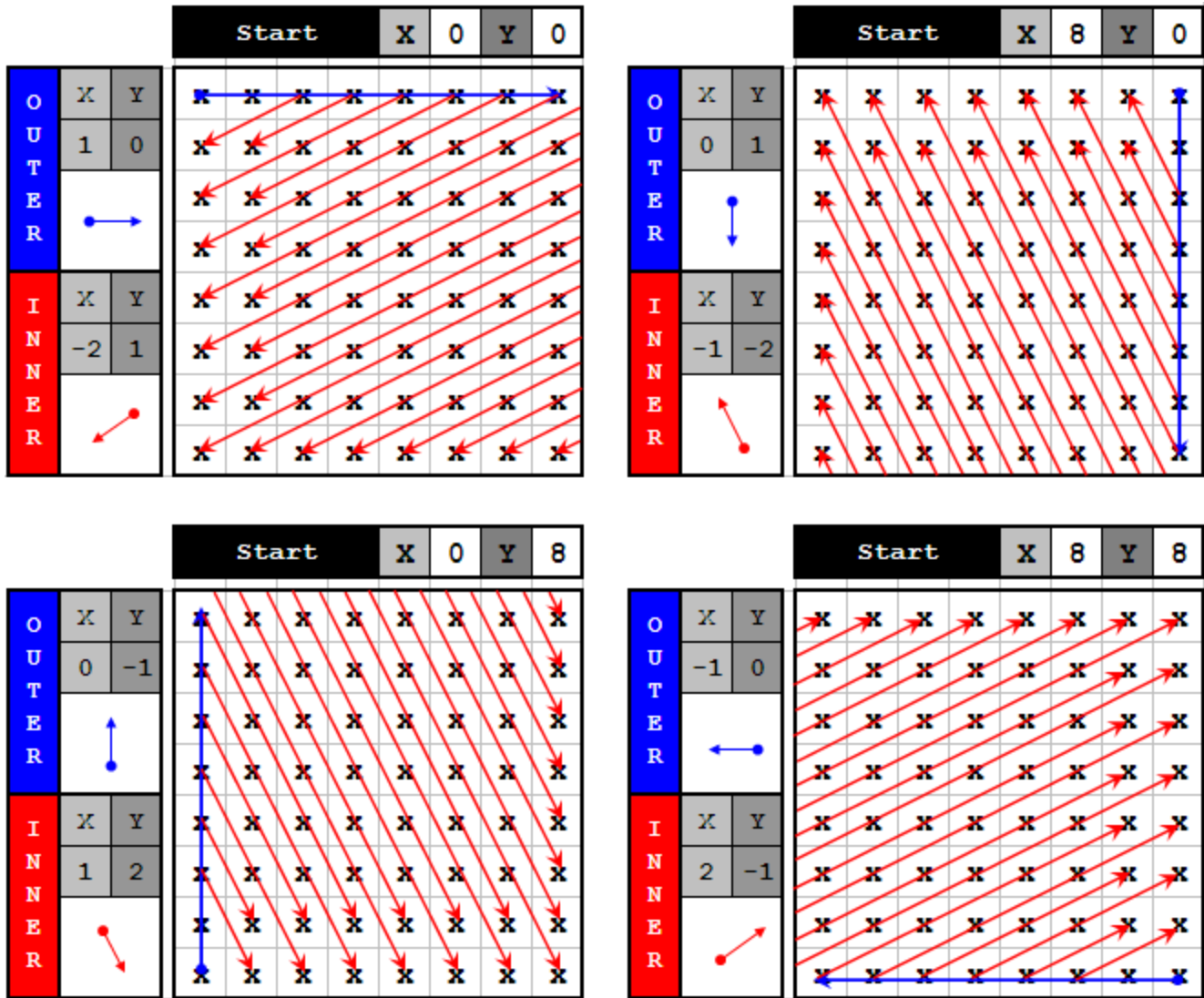


Figure: More Local Loop Walking Patterns



Walker Algorithm Description

The walker algorithm has been tested and optimized in software. A high-level pseudo-code description is given below:

```
Walker(){ //C-Style Pseudo-Code of Walker Algorithm      Load_Inputs_And_Initialize();
  While (Global_Outer_Loop_In_Bounds()){
    Global_Inner_Loop_Intialization();
    While (Global_Inner_Loop_In_Bounds()){
      Local_Block_Boundary_Adjustment();
      Local_Outer_Loop_Initialization();
      While (Local_Outer_Loop_In_Bounds()){
        Local_Middle_Loop_Initialization();
        While (Local_Middle_Steps_Remaining()){
          Local_Inner_Loop_Initialization();
          While (Local_Inner_Loop_Is_Shrinking()){
            Execute();
            Calculate_Next_Local_Inner_X_Y();
          } //End Local Inner Loop
        Calculate_Next_Local_Middle_X_Y();
        } //End Local Middle Loop
      Calculate_Next_Local_Outer_X_Y();
      Calculate_Next_Local_Inverse_Outer_X_Y();
    } //End Local Outer Loop      Calculate_Next_Global_Inner_X_Y();
  } //End Global Inner Loop      Calculate_Next_Global_Outer_X_Y();
} //End Global Outer Loop } //End Walker
```

The above pseudo-code has the following characteristics:

- There are five levels of iteration.
- The highest two levels are called "global" and the lowest three levels are called "local":
 - The global loop contains an outer and an inner loop;
 - The local loop contains an outer, middle, and inner loop;
 - The user defines a bounding box for the global and local resolution;
 - The user also specifies the starting location within each bounding box.
- Each of the five loops has its own persistence:
 - Current position (x,y),
 - Unit step vector (x,y).
- The final output (x,y) is a summation of the global x,y and the local x,y.
- The next (x,y) for a given level can be calculated while the next lower level is still executing. The (x,y) calculation result tests whether the current level needs to execute again once control is returned.

The flow of the global outer and inner loops is as follows:

1. Check a bound condition;
2. Initialize the next level loop;
3. Execute the next level loop;
4. When the next level loop fails its bound condition, calculate the next position for the current loop level and repeat.

Walker Algorithm Flowchart for the Global Loop

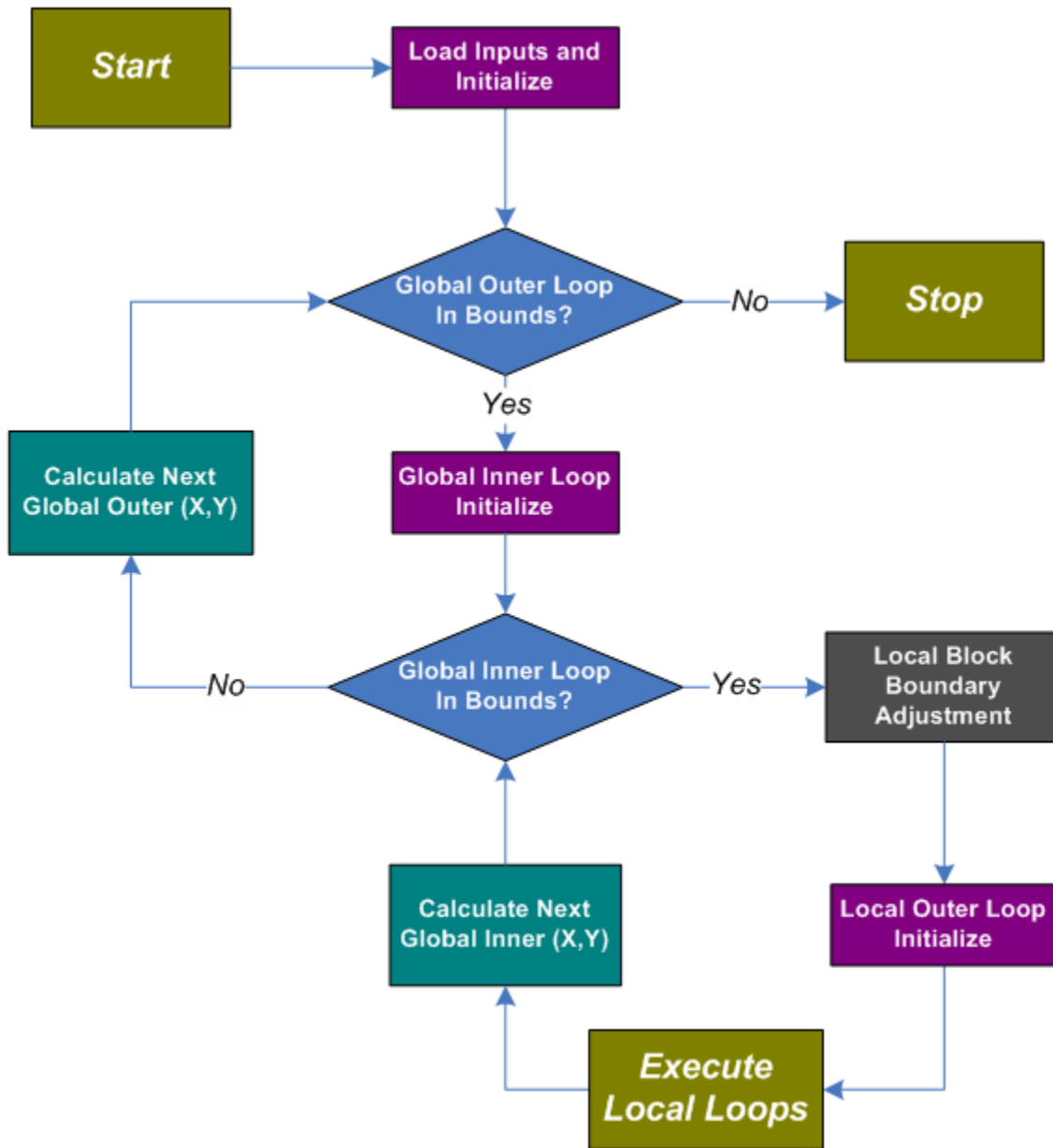


Figure: Walker Algorithm Flowchart for the Global Loop

Note the grey box above labeled, "Local Block Boundary Adjustment." Use this logic to adjust the local block size, when the distance between the current global position and the edge of the image is less than the local resolution. Local starting positions may also be modified by the same logic, if the defined starting position is larger than the new local block size.

The flow of the three local loops does not vary much from the two global loops. The differences are as follows:

- The local middle loop, in addition to a boundary check, also ensures the number of middle steps is less than or equal to the user-defined "number of extra steps."

- The local inner loop only checks to see whether the prior distance between the x,y starting and ending points is greater than their current distance. If this comparison is true, that implies the two inner loops are converging towards each other.
- When the middle loop check fails, both the starting points (local outer) and ending points (local inner) are updated.

Walker Algorithm Flowchart for the Local Loop

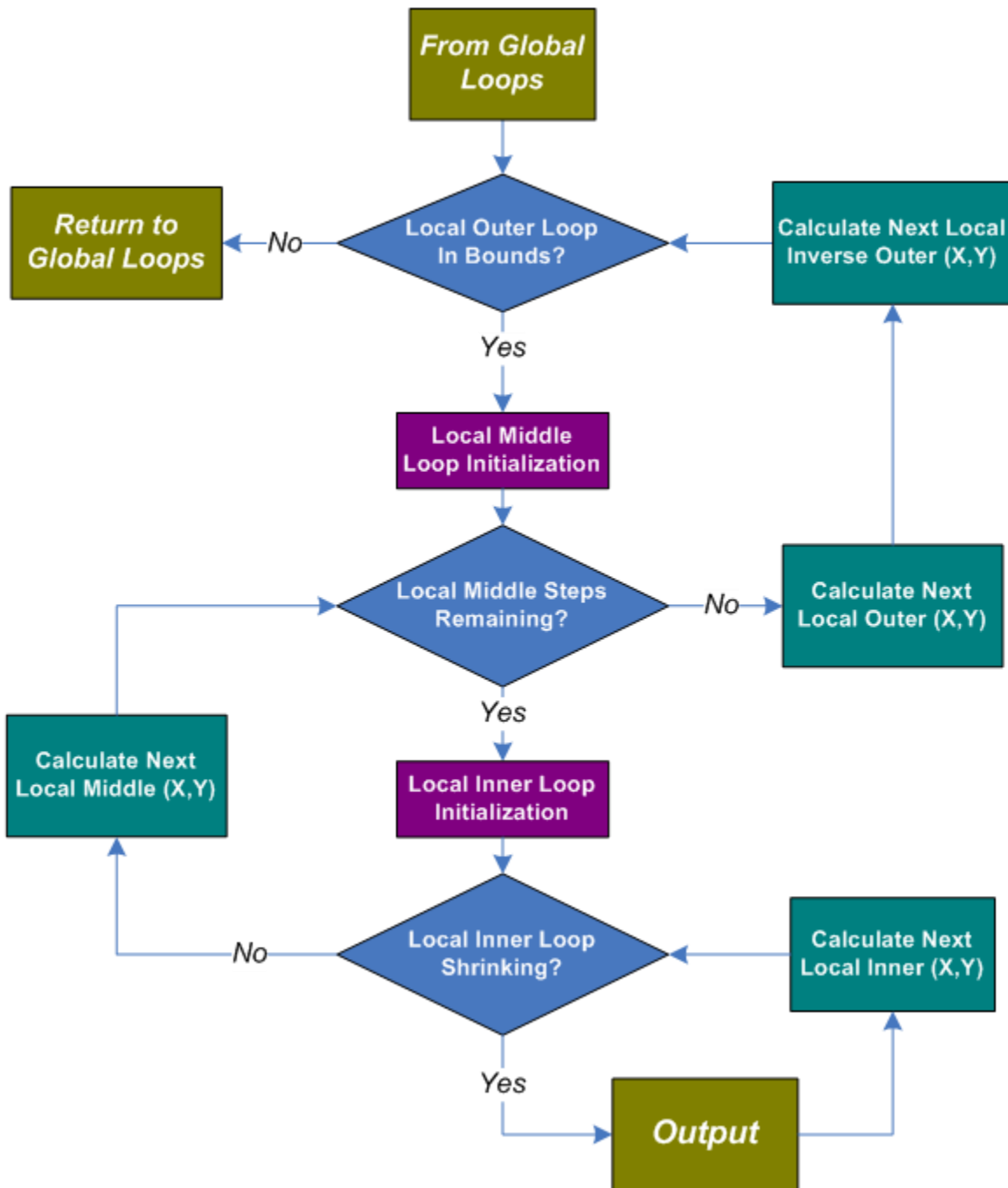


Figure: Walker Algorithm Flowchart for the Local Loop

Flexible Dispatch of Local Loop

A local loop automatically splits between available subslices with adjacent blocks kept next to each other, to improve cache and execution efficiency. A single iteration of the local loop splits into equal segments, one for each currently powered-up subslice. The dispatches for the local loop are done so that one thread from each segment is dispatched before repeating the process. For example: segment 0, thread 0 is followed by segment 1, thread 0 rather than segment 0, thread 1.

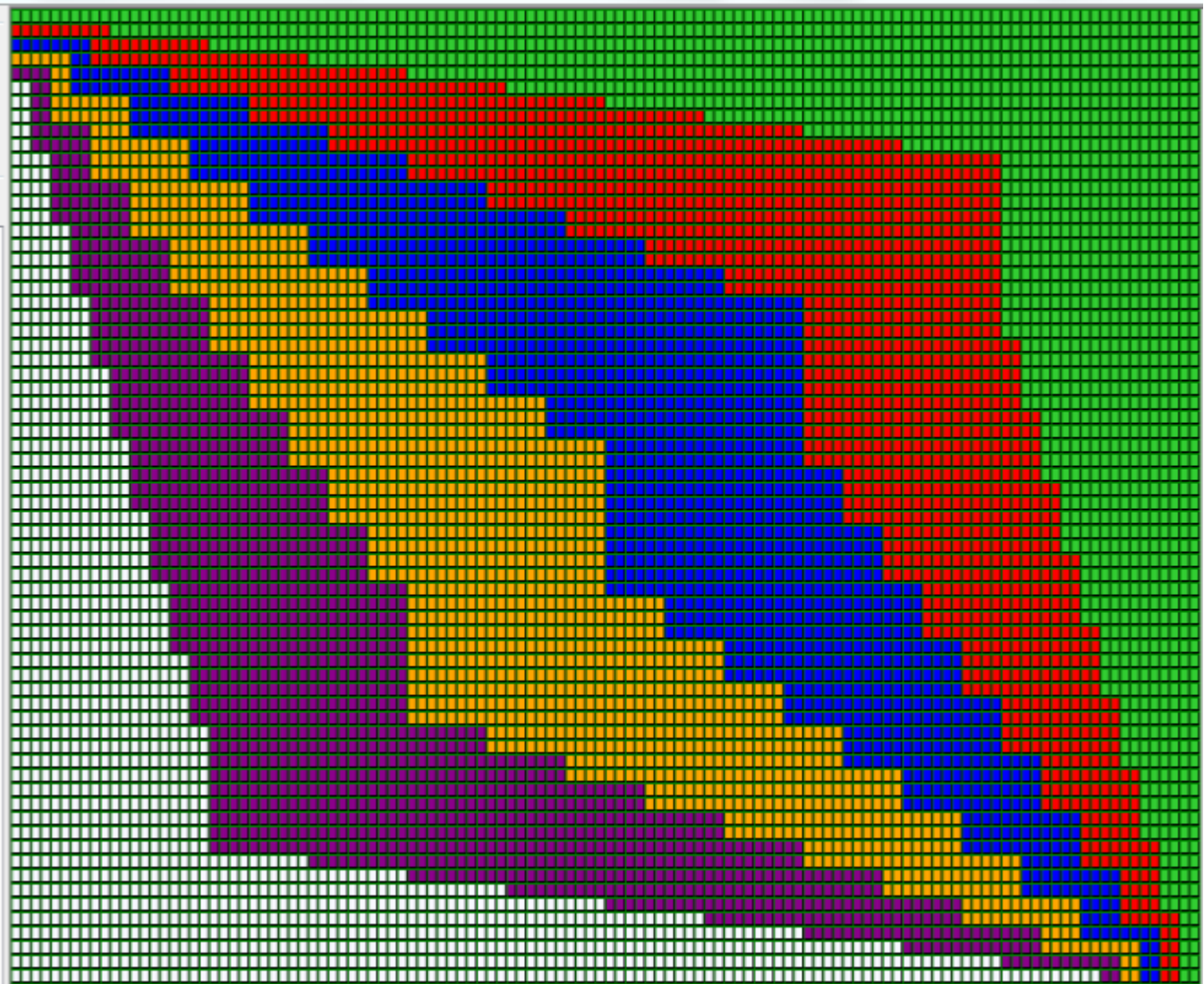


Figure: Splitting a Wavefront Between Six Segments

Each color in the figure above indicates a separate segment dispatched to a specific subslice. In this example, each local loop walks a diagonal line from the lower left to the upper right, while the global loop steps between the lines. Dependencies are to the left and above, which is a typical usage model.

The first few iterations are in the upper right corner and have few blocks dispatched per local loop. Farther down, the length of the local loop gets large enough that the six available subslices are full of threads from each segment running in parallel, and being dispatched in an even manner.

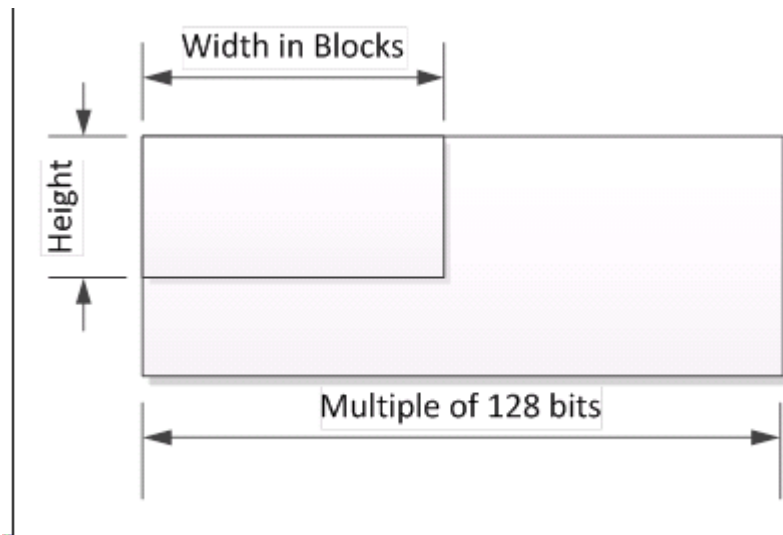
Masked Dispatch

Masked dispatch adds a dispatch mask to the MEDIA_OBJECT_WALKER command, which provides a bit per dispatch to determine whether to drop the dispatch. This allows use of MEDIA_OBJECT_WALKER with irregularly shaped areas. When using the masked dispatch mode, program the walker to walk the surface as a simple rectangle, with X changing in the innermost loop.

CURBE contains the dispatch mask, so a MEDIA_OBJECT_WALKER command that uses the dispatch mask cannot use CURBE in the dispatch payload. CURBE contains a bitmask for which each bit corresponds to the X/Y blockID (not the pixel X/Y). The byte address of each bit is defined as follows:

$$\text{Byte_address} = (Y * \text{pitch} + X) \gg 8$$

Pitch is the number of horizontal blocks in the surface, rounded up to the next power of two bits (e.g. 128, 256, or 512).



The MEDIA_OBJECT_WALKER command allows X and Y to have a maximum of nine bits, which corresponds to a 512 pixels x 512 pixels surface in CURBE. For 16 pixel by 16 pixel blocks, this would represent a total of 8KB x 8KB pixels. Note that in this mode, the walker dispatches in sequence, rather than splitting the local loop into segments that dispatch to individual subslices (as described in "Flexible Dispatch of Local Loop" above).

Thread Dispatch

The following sections cover the thread dispatch process in the GPGPU pipe.

Thread Dispatch Format

The thread dispatch for GPIPE consists of a series of headers and payloads, routed through the dispatch pipeline from the thread spawner unit up until the EU. The thread dispatch consists of transparent and R0 headers and an associated payload that may consist of per-thread and cross-thread data. The transparent header part gets translated as part of the EU state surrounding the threads, becomes the context for a thread while the rest of the payload (including R0 and any optional payload) gets copied to the GRF, and acts as parameters for the thread. Both the transparent and R0 headers are crafted dynamically by hardware based on the interface descriptor.

All threads begin with a transparent header, followed by an R0 header. The format of R0 is different for GPGPU and media threads, as well as for other 3D fixed functions. While the thread spawner generates the headers, the payload is fetched from URB and attached by the dispatch pipeline before dispatch to the EU.

Table: Dispatch Combinations and Associated TSG Phases

Dispatch Type	Dispatch Parameters				TSG Phases (2 or 3 clocks / thread)		
	GPGPU Mode	CURBE Length	Cross Length	Payload Length	Ph-1	Ph-2	Ph-3
Media Thread no CURBE no Payload	0	0	0	0	TH	R0	
Media Thread no CURBE with Payload	0	0	0	>0	TH	R0	Root Handle (DW0)
Media Thread with CURBE no Payload	0	>0	0	0	TH	R0	
Media Thread with CURBE with Payload	0	>0	0	>0	TH	R0	Root Handle (DW0)
GPGPU Thread no Thread ID no CURBE	1	0	0	0	TH	R0	
GPGPU Thread with Thread ID no CURBE	1	>0	0	0	TH	R0	
GPGPU Thread with Thread	1	>0	>0	0	TH	R0	Cross Thread CURBE



ID with CURBE							Handle (DW0)
---------------	--	--	--	--	--	--	--------------

The interface descriptor of a given thread determines whether the CURBE data field is present.

INTERFACE_DESCRIPTOR_DATA

Interface descriptor data is the data structure fetched from the URB, using the interface descriptor data start address. The interface descriptor data serves as the key parameter file to set up the modes of execution, formation, and usage model of threads and thread groups.

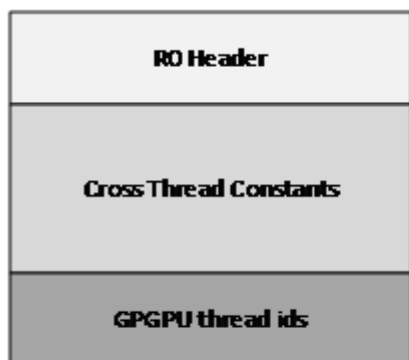
Some of the key parameters controlled by the interface descriptor are as follows:

- Basic thread execution attributes such as the kernel start pointer, floating point mode selection, thread priority definition, handling of de-normalized numbers, and whether the thread can be pre-empted if a mid-thread pre-emption request comes through.
- Number of threads in a GPGPU thread group. This parameter needs to be accurate because it is used in the pre-emption sequence to ensure proper pre-emption. The minimum value is one and the maximum is the number of threads in two subslices for local barrier. For global barrier it may be higher.
- SLM size for a thread group, which is specified in powers of two and minimum increments of 4K blocks up to 64K per subslice.
- Barrier enable and global barrier enable/disable. These parameters define whether there is a barrier associated with a thread group and, if so, can it be local within a subslice or cross-subslice as a global barrier (or both). Setting a global barrier affects performance.
- Constant URB Entry (CURBE) data is a combination of fields. First is the address offset where the constant URB data can be found, followed by a combination of "constant/indirect URB entry read length" and "cross-thread constant data read length." The latter two items define whether there is any payload - and, if so, whether that is per-thread only payload, CURBE only, or both.
- The binding table pointer, and the number of binding table entries that the kernel uses. Also, the sampler state table pointer and sampler count for kernels in use.
- Some exception enables that get loaded directly into the EU CR0 registers.

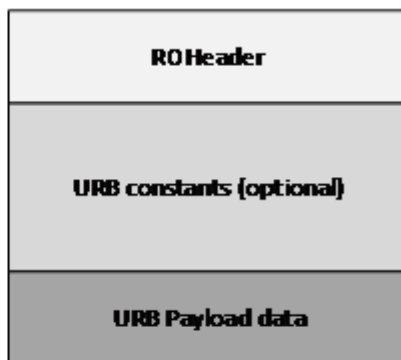
For more details, see the **INTERFACE_DESCRIPTOR_DATA** definition.

R0 Headers

GPGPU and Media R0 headers differ in their organization. Even for Media threads, those with groups attached have a different R0 header than the generic mode R0 header.



GPGPU Payload format



Media Payload format

GPGPU R0 Header

The thread dispatch payload for a GPGPU thread consists of:

An R0 header;

If present, cross-thread CURBE data in R1 and subsequent registers; and

Per-thread payload (e.g. X/Y/Z thread IDs) in the next available registers of the thread - starting in R1 if there is no CURBE data.

DWord	Bits	Description
R0.7	31:0	Thread Group ID Z: This field identifies the Z component of the thread group that this thread belongs to.
R0.6	31:0	Thread Group ID Y: This field identifies the Y component of the thread group that this thread belongs to.
R0.5	31:10	Scratch Space Pointer. Specifies the 1K-byte aligned pointer to the scratch space (used for the GPGPU local memory space). Format = GeneralStateOffset[31:10]
R0.5	9:0	FFTID. This ID is assigned by TS 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	31:5	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	Reserved: MBZ
	3:0	Indicates the stack memory size.

DWord	Bits	Description
		Range = [0,11] indicating [1K bytes, 2M bytes] Programming Note: Exception handling on stack overflow is not supported when GPGPU mid-thread pre-emption is desired.
R0.3	31:5	Sampler State Pointer. Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]
	4	Reserved: MBZ
	3:0	Per Thread Scratch Space. Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two is raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1K bytes, 2M bytes] in powers of two.
R0.2	31	Barrier Enable. This field indicates that a barrier has been allocated for this kernel.
	30:24	BarrierID. This field indicates the barrier that this kernel is associated with. Format: U7 Range = [0,63]
	14:11	Reserved: MBZ
	10	GPGPU Dispatch - This field indicates that the dispatch is from GPGPU_WALKER rather than the various media dispatch instructions.
	3:1	Reserved: MBZ
	0	FreeThread - This bit is set in the first thread of a set of fused threads to help properly track the available threads. In non-fused mode it is set on every thread.
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	31:24	SLM ID. This field encodes whether SLM was allocated for this thread, and if so, the SLM base offset used with SLM operations. Format: U8
	23:17	Reserved: MBZ
	16	FFSID: Fixed Function Stack ID. This field indicates which stack the thread belongs to.
	15:0	URB Handle: This is the URB handle indicating the URB space for use by the thread.

Cross-thread CURBE if present is in R1 and above, followed by the X/Y/Z thread id values for each channel in the thread.

MEDIA R0 Header

The thread payload messages for generic mode and for MEDIA_OBJECT_WALKER are in the same format. The R0 header is the same in both cases, then there are constants from CURBE when CURBE is enabled, followed by inline data when inline is enabled. The inline data block field is the same as in the MEDIA_OBJECT command, with zero-filled partial GRF, and there is no indirect data block field.

The R0 header of the MEDIA_OBJECT_GRPID and MEDIA_OBJECT_WALKER payload with groups enabled is different from the generic mode and MEDIA_OBJECT* payload without groups enabled. This header payload difference is shown in the table below.

Table: Media R0 Header Payload

DWord	Bits	Description
R0.7	31:0	<p>Group ID LSB. This is the LSBs of the Group ID. For MEDIA_OBJECT_GRPID threads this is the entire group id. For MEDIA_OBJECT_WALKER threads the interpretation depends on the Group ID Loop Select:</p> <p>0: No groups, field is 0.</p> <p>1: cat(InnerGlobalCnt[6:0], OuterLocalCnt[9:0], MidLocalCnt[4:0], InnerLocalCnt[9:0]); rest of group id is in R0.2.</p> <p>2: cat(OuterGlobalCnt[6:0], InnerGlobalCnt[9:0], OuterLocalCnt[9:0], MidLocalCnt[4:0]); rest of group id is in R0.2.</p> <p>3: cat(2'b0, OuterGlobalCnt[9:0], InnerGlobalCnt[9:0], OuterLocalCnt[9:0]).</p> <p>4: cat(12'b0, OuterGlobalCnt[9:0], InnerGlobalCnt[9:0]).</p> <p>5: cat(22'b0, OuterGlobalCnt[9:0]).</p>
R0.5	31:10	<p>Scratch Space Pointer. Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled.</p> <p>Format = GeneralStateOffset[31:10]</p>
	9:0	<p>FFTID. This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion.</p>
R0.4	31:5	<p>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.</p> <p>Format = SurfaceStateOffset[31:5]</p>
	4:0	Reserved: MBZ
R0.3	31:5	<p>Sampler State Pointer. Specifies the 32-byte aligned pointer to the sampler state table.</p> <p>Format = GeneralStateOffset[31:5]</p>
	4	Reserved: MBZ
	3:0	<p>Per Thread Scratch Space. Specifies the amount of scratch space, in 1K-byte quantities, allowed to be used by this thread. The value specifies the power to which two is raised to determine the amount of scratch space.</p> <p>Format = U4</p> <p>Range = [0,11] indicating [1K bytes, 2M bytes] in powers of two</p>
R0.2	31	<p>Barrier Enable. This field indicates that a barrier has been allocated for this kernel.</p>
	30:24	<p>BarrierID. This field indicates the barrier that this kernel is associated with.</p> <p>Format: U7</p> <p>Range = [0,63]</p>

DWord	Bits	Description
	23:11	Group ID MSB. This field is the MSBs of the Group ID. These bits are 0 for MEDIA_OBJECT_GRPID. For MEDIA_OBJECT_WALKER threads the interpretation depends on the Group ID Loop Select: 0: No groups, field is 0. 1: cat(OuterGlobalCnt[9:0], InnerGlobalCnt[9:7]). 2: cat(10'b0, OuterGlobalCnt[9:7]). 3-5: All zero.
	10	Reserved: MBZ
	9:4	Interface Descriptor Offset. This field specifies the offset from the interface descriptor base pointer to the interface descriptor that is applied to this object. It is specified in units of interface descriptors. Format = U6
	3:0	Reserved: MBZ
R0.1	31:24	Block Color. This field is the 8-bit Media block Color value. It specifies which dependency color the current thread belongs to Format = U8
	23	Reserved: MBZ
	22:12	Block Y. This field provides the Y media block coordinate of current thread. Format = U11
	11	Reserved: MBZ
	10:0	Block X. This field provides the X media block coordinate of current thread. Format = U11
R0.0	31:24	SLM ID. This field encodes whether SLM was allocated for this thread, and if so, the SLM base offset used with SLM operations.
	23:17	Reserved: MBZ
	16	FFSID: Fixed Function Stack ID. This field indicates which stack the thread belongs to.
	15:0	URB Handle. This is the URB handle where indicating the URB space for use by the root thread and its children.

Thread Tracking and Synchronization

You must track a dispatched thread for the End Of Thread (EOT) lifecycle. The tracking process in TSG creates renewable capacity in the EU farms, and renews thread resources by releasing them once the thread lifecycle finishes. For GPGPU, the synchronization between threads is done through barrier; in Media, the synchronization is done through the scoreboard.

Thread Synchronization Monitors

Thread Synchronization Monitors (TSM) are a hardware feature used for inter-thread communication. It can be used to implement a software scoreboard or other synchronization method with a higher performance than using polling. Unlike barriers, TSM allows communication across all thread in the entire machine, rather than just the threads in a thread group.

TSM uses messages to the Message Gateway to monitor and signal events. The messages used are:

1) MonitorEvent - this message indicates to the Gateway that a particular Event ID is of interest to the sending thread and should be monitored. When the corresponding event occurs, it is recorded in the Gateway but no message is sent until a WaitForEvent message is sent.

2) WaitForEvent - this message tells the Gateway that the thread is ready to go to sleep until the event or a timeout occurs. After sending this message the thread should read the writeback register to allow the EU to switch to other threads until the event occurs and the dependency is cleared. The message payload specifies how long the Gateway should wait before signaling a timeout, and the writeback register specified gets a value of 0 if a timeout occurs or 1 if the event occurs. Note that a pre-emption will always cause a timeout.

3) SignalEvent - this message is broadcast to every Gateway (including the source Gateway). The event is recorded for each thread that has a MonitorEvent with a matching event ID, while a message is sent to each thread which has a WaitForEvent outstanding. If the WaitForEvent occurs after the SignalEvent then the message will go out immediately upon receiving the WaitForEvent. After the message goes out thread will still be monitoring that event, so it can be reused as long as care is taken that every thread gets the same number of events - if an event happens twice while a thread is merely monitoring before WaitForEvent it will only see the event once.

4) MonitorNoEvent - this message disables the monitoring of events for this thread. This is also done when the thread exits with the EOT (End of Thread) message.

See the Message Gateway section of the Bspec for more details of these messages.

Here is an example of using TSM for a software scoreboard which is used to determine if macroblocks that the current thread is dependent on have completed:

```
For (all dependencies for this macroblock) {
    MonitorEvent(scoreboard[dx][dy]); // Start monitor before checking dependency bit
    While (!atomic_read(scoreboard[dx][dy])) {
        WaitForEvent(EventID(dx,dy), timeout); // EventID determined from dx & dy
    }
}
```



```
// MonitorNoEvent is not needed here
DoBlock(); // All normal processing of block
Atomic_write(scoreboard[x][y], 1); // Set the dependency bit for following blocks
SignalEvent(EventID(x,y));
```

Barriers and Shared Local Memory

Barriers and Shared Local Memory (SLM) offer advantages for both general purpose and media applications. Barriers can synchronize between GPGPU and media threads more efficiently than atomics can do, while SLM can be used to share data between tightly associated threads.

Barriers and SLM can also be used with a more generalized walker, to add the ability to use the scoreboard at the same time. Implementation requires an identifying number similar to the thread group ID that can be used to track and free resources. The MEDIA_OBJECT_WALKER can use various loop counts to define the group ID.

For flexibility, you can specify which loop counts form the group ID, and which ones count the threads inside each group. If cross-slice barriers are disabled, the walker must ensure that all threads in a group are dispatched to a single subslice so that the barrier and SLM are available to all group members. Make sure that the number of threads generated per group is no larger than the number of threads available in a subslice.

MEDIA_OBJECT_WALKER adds the X/Y values for the various loops together to produce a single X/Y point. Since some walking patterns have overlapping X/Y values, the combined X/Y point cannot be used for the global ID; instead, execution counts are concatenated together to produce an ID number. (An execution count is a tally of how many times each loop is executed.) The GPGPU thread group ID is a 96-bit number, so it easily contains the 49 bits created by all the execution counts.

The media walker has five nested loops for producing the X/Y point. In addition, there is an innermost color loop defined as follows:

1. Color loop - 4 bits
2. Inner local loop execution count - 10 bits
3. Mid local loop execution count - 5 bits
4. Outer local loop execution count - 10 bits
5. Inner global loop execution count - 10 bits
6. Outer global loop execution count - 10 bits

A media walker parameter specifies the global ID bits by identifying at what point to switch between the group ID and per-thread ID. Unused loops always have execution counts of zero. The media walker forms group IDs by concatenating execution counts that are enabled in the Group ID Loop Select to create the LSBs of the IDs, and sets any unused MSBs to zero.

Barriers and SLM Example:

If a MEDIA_OBJECT_WALKER specifies that the outer local loop count and above form the group ID, then every iteration of the color, inner local, and mid local dispatches a thread with the same group ID. TSG

allocates a shared barrier and SLM (as specified in the interface descriptor), and ensures that all threads go to the same subslice. If no global barriers are used, the number of threads must fit on a single subslice; if global barriers are used, the number of threads must fit on the system.

The group ID for this example is formed by concatenating the outer global exec count, inner global exec count, and outer local exec count. When the group ID increments, the TSG allocates a new barrier and SLM and pick an available subslice.

Media-GPGPU Thread EOT Message

The thread EOT message is issued to the Gateway unit by a GPGPU or Media thread running on an EU. This message contains only one 8-DWord register which is intended to be a direct copy of R0 in the thread payload. It indicates that the thread is terminating, and the thread spawner should dereference the barrier and SLM resources that were optionally allocated with the thread when it was dispatched.

Message Descriptor

Thread EOT Message Descriptor

Message Payload

DWord	Bits	Description
M0.5	31:10	Ignored.
	9:0	Ignored.
M0.4	31:0	Ignored.
M0.3	31:0	Ignored.
M0.2	31	Barrier Enable. This field indicates whether a barrier was allocated for this thread. Format: Enable
	30:24	BarrierID. When Barrier Enable is set, this field identifies which barrier was allocated for this thread. Format: U7
	23:0	Ignored.
M0.1	31:0	Ignored.
M0.0	31:24	SLM ID. This field encodes whether SLM was allocated for this thread, and if so, the SLM base offset used with SLM operations. Format: U8
	23:0	Ignored



Context Switch for GPGPU and Media

In the ExecList scheduling mode, software triggers preemption by submitting a new pending execlist to the ExecList Submit Port (ELSP). On detecting the availability of the new pending execlist, hardware triggers preemption on a preemptable command. Following successful preemption, the newly submitted execlist undergoes a context switch. As part of the context switch, the Logical Ring Context Address (LRCA) saves the preempted context state. The context state contains details for hardware to resume execution of a resubmitted context from the point where the context was preempted.

The GPGPU Pipe supports several types of preemption boundaries:

GPGPU Pipe Preemption Boundaries

Preemption Boundary	Description
Command (coarsest granularity)	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 in the middle of a GPGPU Walker or Media Walker command. After all previously dispatched threadgroups have completed, 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 (finest granularity)	The context is preempted in the middle of a GPGPU Walker command. After all previously dispatched threadgroups have started running in an EU, each running thread is preempted at the next interruptible EU instruction, all per-thread data is saved, the current walker position is saved along with the current context state. When the context is resumed, all the per-thread data is restored, the threads are re-started, and the current walker command is executed to re-start at the next threadgroup dispatch in the walker.

If preemption is signaled and the Preemption Control specifies a boundary not supported the currently running command or command streamer, then the preemption will occur on the boundary supported by that command and command streamer.

If a particular sequence of commands within a context needs to be protected from preemption, then it must be explicitly put between the MI_ARB_OFF/MI_ARB_ON command pair.

Preemption Boundaries supported by GPGPU Pipe Commands and Command Streamers

Recognized Preemption Boundaries	Current GPGPU Pipe Command	Which Render Engine Command Streamer is running the Command
<ul style="list-style-type: none"> Mid-thread Threadgroup Command 	GPGPU_WALKER	RCS
<ul style="list-style-type: none"> Threadgroup Command 	GPGPU_WALKER	CCS

Recognized Preemption Boundaries	Current GPGPU Pipe Command	Which Render Engine Command Streamer is running the Command
<ul style="list-style-type: none"> • Threadgroup • Command 	MEDIA_OBJECT_WALKER	RCS
<ul style="list-style-type: none"> • Command 	All other GPGPU Pipe Commands, including: <ul style="list-style-type: none"> • MEDIA_VFE_STATE, • MEDIA_STATE_FLUSH, • MEDIA_LOAD_INTERFACE_DESCRIPTOR • MEDIA_CURBE_LOAD Any MI_ Command when MI_ARB_ON, including: <ul style="list-style-type: none"> • PIPE_CONTROL • PIPELINE_SELECT • MI_ARB_CHECK • MI_WAIT_FOR_EVENT • MI_SEMAPHORE_WAIT 	RCS

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).

Colssue/Multi Issue:

Multiple execution pipes operate simultaneously, scheduler selects an execution pipe based on their availability and capability.

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.

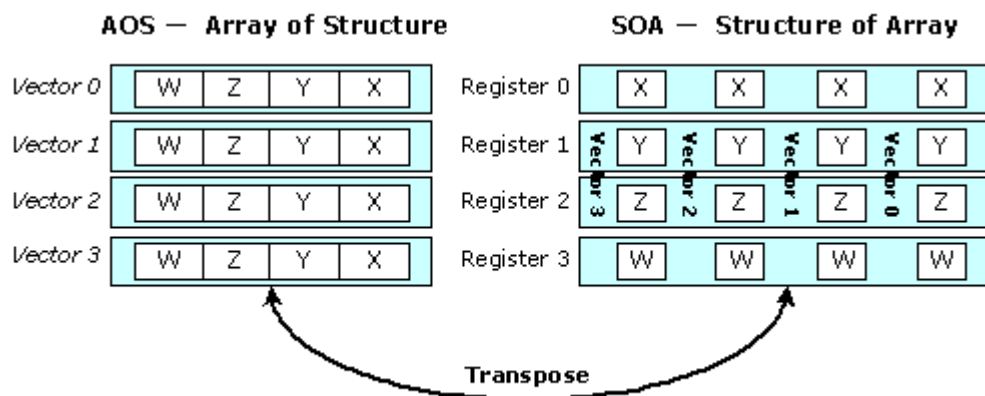
Primary Usage Models

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



B6890-01

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 $SIMD_{m \times n}$, 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 stored in 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,

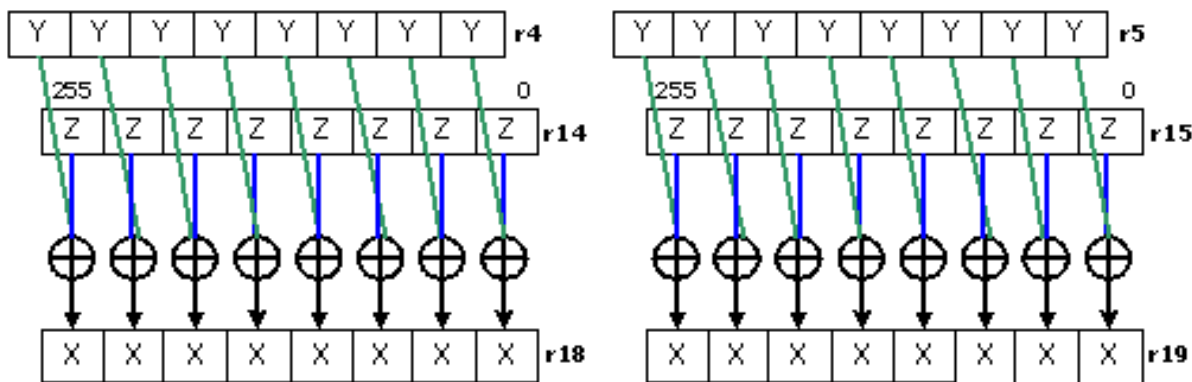
the example 3D graphics API Shader instruction can be translated into the following three instructions:

```
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



```
Add (16) r18<1>:f r4<8;8,1>:f r14<8;8,1>:f {Compr} // dst.x=src0.y+src1.z
```

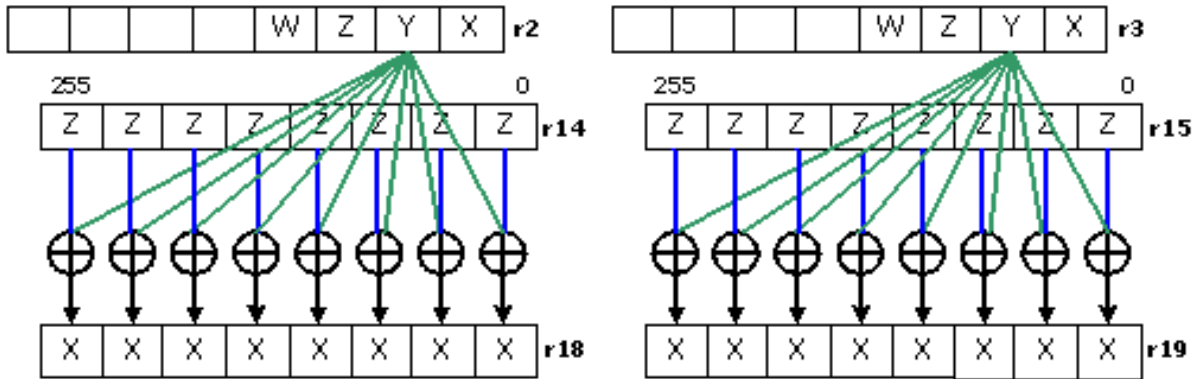
B.6894-01

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



```
Add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr} // dst.x=src0.y+src1.z
```

B6895-01

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:

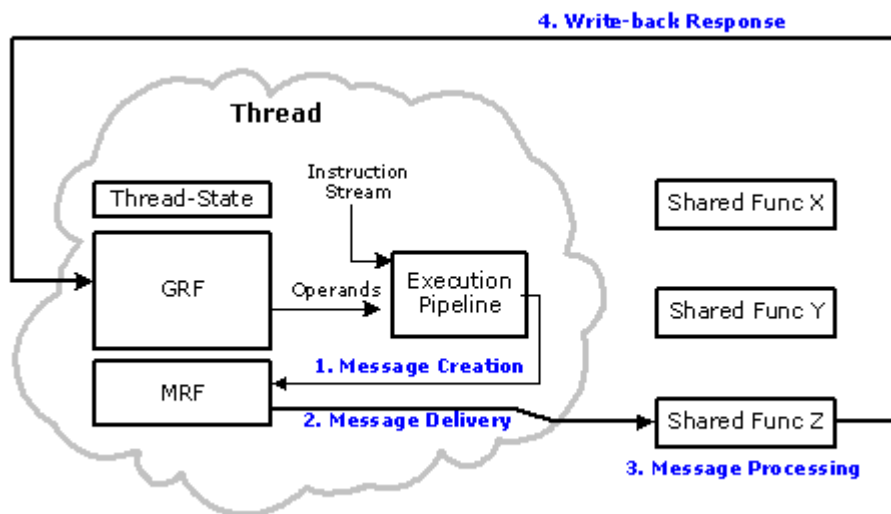
- **Message Payload.**
- Associated ("sideband") information provided by:

- **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 order 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 <u>not</u> supporting EOT messages
PixelPort, URB, ThreadSpawner	DataPort

Target Shared Functions supporting EOT messages	Target Shared Functions <u>not</u> 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

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 treats 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 an 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.
Incorrect response length	<p>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.</p> <p>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 hang and result in an eventual time-out.</p>
Improper use of End-Of-Thread (EOT)	<p>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.</p> <p>The 'send' instruction specifies that EOT is only recognized if the <desc> field of the instruction is an 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.</p>
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 some times 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	Fixed allocation of 128 registers	128 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	<i>null</i>	Null register
0001b	<i>a0.#</i>	Address register
0010b	<i>acc#</i>	Accumulator register
0011b	<i>f#.#</i>	Flag register
0100b	<i>ce#</i>	Channel Enable register
0101b	<i>msg#</i>	Message Control Register
0110b	<i>sp</i>	Stack Pointer Register
0111b	<i>sr0.#</i>	State register
1000b	<i>cr0.#</i>	Control register

Register Type	Register Name	Description
1001b	<i>n#</i>	Notification Count register
1010b	<i>ip</i>	Instruction Pointer register
1011b	<i>tdr</i>	Thread Dependency register
1100b	<i>tm0</i>	Pause register
1101b	<i>fc#.#</i>	Flow Control register
1110b	<i>Reserved</i>	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

Attribute	Value
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.	<p>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.</p> <p>00000b = a0.0:uw</p> <p>00010b = a0.1:uw</p> <p>00100b = a0.2:uw</p> <p>00110b = a0.3:uw</p> <p>01000b = a0.4:uw</p> <p>01010b = a0.5:uw</p> <p>01100b = a0.6:uw</p> <p>01110b = a0.7:uw</p> <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> <p>10000b = a0.8:uw</p> <p>10010b = a0.9:uw</p> <p>10100b = a0.10:uw</p> <p>10110b = a0.11:uw</p> <p>11000b = a0.12:uw</p> <p>11010b = a0.13:uw</p> <p>11100b = a0.14:uw</p> <p>11110b = a0.15:uw</p> </div> <p>All other encodings are reserved.</p> <p>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.</p>

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 <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <i>send</i> 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 <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <i>send</i> instruction, it provides the lower 16 bits of <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 <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <i>send</i> 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 <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For a <i>send</i> 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
<p>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.</p>	

Programming Note	
Context:	ARF Registers
<p>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.</p>	

Accumulator Registers

Accumulator Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	2+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-1001b = acc0-acc9 All other encodings are reserved.	0 : Lower half 1 : Upper half	Reserved: MBZ

- 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.

Description
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}
```

Description
It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. " <i>add (8) acc1:f acc0:f</i> "). The result of such an instruction is unpredictable.
Accumulator registers may be accessed explicitly as src0 operands only.

- Swizzling is not allowed 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 1FFFFFFFFeh. 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	4	64	When accumulator is used for Double Float, it has the exact same precision as any GRF register.
F	acc0/acc1	8	32	When accumulator is used for Float, it has the exact same precision as any GRF register.
HF	acc0/acc1	16	16	When accumulator is used for Half Float, it has the exact same precision as any GRF register.
Q	N/A	N/A	N/A	Not supported data type.
D (UD)	acc0	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 <i>mul</i> , <i>mach</i> , and <i>shr</i> (or <i>mov</i>). Writing to acc1 may corrupt this result.
W (UW)	acc0	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 may corrupt this result.
B (UB)	N/A	N/A	N/A	Not supported data type.

Math Macro Extended Accumulators

Special Accumulators acc2-acc9



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.

Example:
<p>These registers may be accessed explicitly by a <code>mov</code> operation, with no source modifiers, condition modifiers, or saturation. When accessed explicitly, the datatype must be D. On reads, the low 2 bits of each DWord are valid data. The other bits are undefined. On writes, the low two bits are updated, and other bits are dropped.</p> <pre>// Move 256 bits from acc8 to r10. Just low two bits of each DWord are valid: mov (8) r10:ud acc8:ud // Move 256 bits from r10 to acc8. Just low two bits of each DWord are updated: mov (8) acc8:ud r10:ud</pre>

- 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.
- Implicit access across accumulator registers is required for each source operand for these macro instructions. The `SubRegNum` bits in the instruction are used to implicitly address the different accumulators. The `noacc` value is specified when no write to accumulator is required; think of it as a null.

SpecialAcc[3:0] Encoding	Accumulator Register
0000b	acc2
0001b	acc3
0010b	acc4
0011b	acc5
0100b	acc6
0101b	acc7
0110b	acc8
0111b	acc9
1000b	noacc

- The MSB bits [4:1] of the source `SubRegNum` are used to specify the `SpecialAcc[3:0]`, similarly the destination `SubRegNum` bits are used to specify `SpecialAcc[3:2]` with the lower bits `SpecialAcc[1:0]` directly specified in the instruction.

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	00000b = fn.0:uw
0001b = f1:ud	00010b = fn.1:uw
Other encodings are reserved.	Other encodings are reserved.

Description
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	00000b = ce:ud .
All other encodings are reserved.	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.

Message Control Registers

Message Control Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0101b
Number of Registers:	1
Default Value:	None
Normal Access:	RO/PW
Elements:	1
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control?	No
Indexable?	No

Register and SubRegister Numbers for Message Control Registers

RegNum[3:0]	SubRegNum[4:0]
0000b = msg0 All other encodings are reserved.	00000b = msg0.0:ud All other encodings are reserved

These are specific control registers used to track messaging. These may be saved and restored by the kernel only when a thread is in the context save/restore mode. Access of these registers otherwise, will result in undeterministic behavior.

Each thread has 1 registers. The granularity of access is always one full register. These registers must be accessed with a MOV with no predication, src modifiers or conditional modifiers. They MUST be accessed in direct addressing more only.

DWord	Bits	Description
0 (msg0:ud)	31:22	Reserved. MBZ.

Stack Pointer Register

Stack Pointer Register Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0110b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	2
Element Size:	64 bits
Element Type:	UD
Access Granularity:	DWord
Write Mask Granularity:	DWord
SecHalf Control	No
Indexable	No

The stack pointer register can be accessed as a unsigned DWord or QWord integer. It consists of a read-write register, containing the current stack pointer (sp.0), which is relative to the Generate State Base Address, and a read-only stack limit register (sp.1). The stack pointer is inserted into the message header when data is stored into scratch space as a stack. The stack pointer is managed by software. If the stack pointer exceeds the limit or the space allocated, an exception is triggered. See the Stack Pointer Exception in the Exceptions Section. Writes to the stack pointer must use the {Switch} ThreadCtrl option.

Register and Subregister Numbers for SP Register

RegNum[3:0]	SubRegNum[4:0]
0000b = sp All other encodings are reserved.	00000b = sp.0:uq . 01000b = sp.1:uq All other encodings are reserved.

SP Register Fields

DWord	Bits	Description
0..1 (sp.0:ud)	63:48	Reserved. MBZ.
	47:0	<p>sp.0. Specifies the current stack pointer. This pointer is relative to the General State Base Address. This register is initialized at thread load to the top of the per thread Scratch Space. The register is R/W.</p> <p>$sp = [scratch\ space\ pointer] + [scratch\ space] - 1$</p> <p>Alternatively, this register may be updated by Software to any flat address space. In such cases, the stack is NOT relative to the General State Base Address. Software must ensure that the address is</p>

DWord	Bits	Description
		exclusive for the thread.
2..3 (sp.1:ud)	63:48	Reserved. MBZ.
	47:0	<p>sp.1. Specifies the upper limit for the stack pointer. This pointer is relative to the General State Base Address. This register is initialized at thread load to the limit allocated for stack in the state. See the GPGPU Thread Payload description for details. The register is R/W.</p> <p><i>sp_limit = [scratch space pointer] + [stack space limit]</i></p> <p>Alternatively, this register may be updated by software, similar to the sp register. In such cases, software is responsible for allocating the right thread stack pointer limit.</p>

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. 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. 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.
	19:14	Reserved. MBZ.
13:4	[13:11] Slice ID. [10:9] Dual-SubSlice ID [8] SubSlice ID.	

DWord	Bits	Description													
		[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. Initialized by hardware. Read-only.													
1 (sr0.1:ud)	31:21	FFTID (Fixed Function Thread ID). There is no connection between this thread ID, assigned in fixed functions, and the TID assigned in the EUs.													
	15:7	Reserved. MBZ													
	5:0	<p>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:</p> <p>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.</p> <table border="1"> <thead> <tr> <th>Bits</th> <th>Definition</th> </tr> </thead> <tbody> <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> </tbody> </table> <p>Initialized to 0. Read/Write</p>	Bits	Definition	5	Reserved	4	Inexact Exception	3	Overflow	2	Underflow	1	Divide by Zero	0
Bits	Definition														
5	Reserved														
4	Inexact Exception														
3	Overflow														
2	Underflow														
1	Divide by Zero														
0	Invalid Operation														
2 (sr0.2:ud)	31:0	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.													
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.													

DWord	Bits	Description
		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 master 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 Master 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 = <i>cr0</i>	00000b = cr0.0:ud . It contains general thread control fields.
All other encodings are reserved.	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	<p>Master Exception State and Control. This bit is the master 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.</p> <p>0 = The thread is in normal state. 1 = The thread is in exception state.</p> <p>Initialized by hardware. Read/Clear.</p>
	30:16	Reserved. MBZ.

DWord	Bits	Description
	14	<p>Thread preemption disable: This field specifies whether the thread is allowed to stop in middle on receiving mid-thread preemption request.</p> <p>0 = Thread is preempted on receiving preemption indication.</p> <p>1 = Thread is preempted only in case of page-fault.</p>
	13	Reserved. MBZ.
	12	<p>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.</p> <p>This bit must be set for IEEE compliant float to integer conversion operation.</p> <p>0= The result of float to integer conversion operation is with RTZ rounding mode.</p> <p>1= The result of float to integer conversion operation is with rounding mode programmed in cr0.0[5:4].</p> <p>Initialized to 0. Read/Write.</p>
	11	<p>IEEE MinMax. This bit determines how SNAN is handled in min/max operations. This bit is initialized to 0 during Thread Dispatch.</p> <p>This bit must be set for IEEE compliant min/max operation.</p> <p>0 = The result of min/max is a non-SNAN source.</p> <p>1 = The result of min/max is the SNAN source.</p> <p>Initialized to 0. Read/Write.</p>
	10	<p>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.</p> <p>0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign.</p> <p>1 = Allow denorm source values and denorm results.</p> <p>Initialized to 0. Read/Write.</p>
	9	<p>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 <i>not</i> transfer control to any handler.</p> <p>0 = IEEE Exception flags are NOT trapped.</p> <p>1 = IEEE Exception flags are trapped.</p> <p>Initialized to 0. Read/Write.</p>
	7	<p>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</p>

DWord	Bits	Description
		<p>mode, which always flushes denorms. This bit is initialized by Thread Dispatch.</p> <p>0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign.</p> <p>1 = Allow denorm source values and denorm results.</p> <p>Initialized at dispatch. Read/Write.</p>
	6	<p>Double Precision Denorm Mode. This bit determines how denormal numbers are handled for the DF (Double Float) type. It is initialized by Thread Dispatch.</p> <p>0 = Flush denorms to zero when reading source operands and flush denorm calculation results to zero. Denorm flushing preserves sign.</p> <p>1 = Allow denorm source values and denorm results.</p> <p>Initialized to 0. Read/Write.</p>
	5:4	<p>Rounding Mode. This field specifies the FPU rounding mode. It is initialized by Thread Dispatch.</p> <p>00b = Round to Nearest or Even (RTNE)</p> <p>01b = Round Up, toward +inf (RU)</p> <p>10b = Round Down, toward -inf (RD)</p> <p>11b = Round Toward Zero (RTZ)</p> <p>Initialized at dispatch. Read/Write.</p>
	3	<p>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.</p> <p>0: Use Dispatch Mask (DMASK)</p> <p>1: Use Vector Mask (VMASK)</p> <p>Initialized at dispatch. Read/Write.</p>
	2	<p>Single Program Flow (SPF). Specifies whether the thread has a single program flow (SIMD_nx_m with m = 1) or multiple program flows (SIMD_nx_m 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.</p> <p>0: Multiple Program Flows</p> <p>1: Single Program Flow</p> <p>Programming Restrictions:</p> <p>Only H1/Q1/N1 are allowed in SPF mode.</p> <p>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.</p> <p>SPF mode must be set to 0 in Fused threads.</p>

DWord	Bits	Description
		Initialized at dispatch. Read/Write.
	1	<p>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.</p> <p>0: Enable accumulator update. 1: Disable accumulator update.</p> <p>Usage Notes:</p> <p>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 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.</p> <p>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.</p> <p>Initialized to 0. Read/Write.</p>
	0	<p>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. 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.</p> <p>0 = IEEE floating-point mode for the F (Float) type. 1 = ALT (alternative) floating-point mode for the F (Float) type.</p> <p>Initialized at dispatch. Read/Write.</p>
1	30	<p>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.</p> <p>This bit may be set or cleared by software. This bit is initialized to 0.</p> <p>Initialized by hardware. Read/Clear.</p>
	29	<p>Software Exception Control. This bit is the control bit for software exceptions. Setting this bit to</p>

DWord	Bits	Description
		<p>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.</p> <p>This bit may be set or cleared by software. This bit is initialized to 0.</p> <p>Initialized to 0. Read/Write.</p>
	28	<p>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.</p> <p>Initialized to 0. Read/Clear.</p>
	27	<p>Stack Overflow Exception Status. This bit when set, indicates a stack overflow exception. The exception handler routine normally does not return back to the application thread upon a stack overflow 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.</p> <p>This bit is initialized to 0.</p>
	25	<p>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.</p>
	24	<p>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.</p>
	23:16	<p>Reserved. MBZ.</p>
	13	<p>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.</p> <p>This bit is initialized by the Thread Dispatcher.</p> <p>Format = ENABLED:</p> <p>0: Disabled</p> <p>1: Enabled</p> <p>Initialized at dispatch. Read/Write.</p>
	12	<p>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.</p> <p>This bit is initialized by the Thread Dispatcher.</p>

DWord	Bits	Description								
		<p>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.</p> <p>Format = ENABLED:</p> <p>0: Disabled</p> <p>1: Enabled</p> <p>Initialized at dispatch. Read/Write.</p>								
	11	<p>Stack Overflow Exception Enable. This bit specifies whether the stack overflow exception is enabled or not. The stack overflow exception includes an overflow or an underflow in the stack space allocated for the thread.</p> <p>This bit is initialized by the Thread Dispatcher.</p> <p>Software should normally assign this bit in the interface descriptor.</p> <p>Format = ENABLED:</p> <p>0: Disabled</p> <p>1: Enabled</p> <p>Initialized at dispatch. Read/Write.</p>								
	10:0	Reserved. MBZ.								
2 (cr0.2:ud)	31:3	<p>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 Master 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.</p> <table border="1" data-bbox="344 1346 1203 1526"> <thead> <tr> <th>Exception Type</th> <th>AIP Value</th> </tr> </thead> <tbody> <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> </tbody> </table> <p>(1) External Halt exception is asynchronous and not associated with an instruction.</p> <p>When the System Routine changes the Master 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.</p> <p>Format = U32[31:3]</p> <p>Not Initialized. Read/Write.</p>	Exception Type	AIP Value	External Halt	N/A ⁽¹⁾	Software Exception	IP + 1	Illegal Opcode	IP
Exception Type	AIP Value									
External Halt	N/A ⁽¹⁾									
Software Exception	IP + 1									
Illegal Opcode	IP									
	2:0	Reserved. MBZ.								

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	00000b = n0.0:ud
All other encodings are reserved.	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.

Implementation Restriction: The notification registers are initialized to 0 after hardware/software reset. However, these registers are not reset at thread dispatch time.

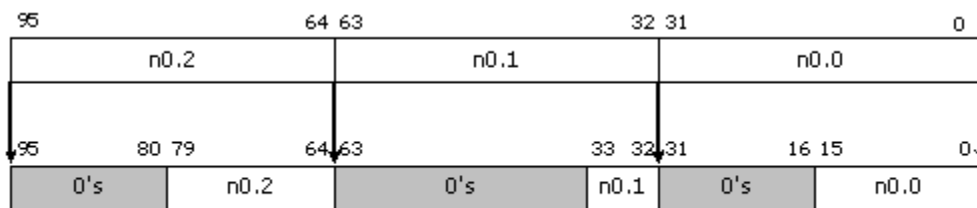
Notification Register 0 Fields

Dword	Bits	Description
0	31:1	Reserved. MBZ.
	0	<p>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.</p> <p>Format: U1</p> <p>Initiallized to 0. Read/PWrite.</p>

Notification Register 1 Fields

DWord	Bits	Description
0	31:1	Reserved. MBZ.
	0	<p>Host to Thread Notification. This register is used by the sync.host instruction for host-to-thread synchronization via MMIO registers.</p> <p>Format: U1</p> <p>Initiallized to 0. Read/PWrite.</p>

Format of the Notification Register



B.6898-01

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
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	00000b = ip:ud
All other encodings are reserved.	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



Attribute	Value
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 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 not commence until the write to the 'Pause' register actually retires.

Programming Note	
Context:	Pause Register - Workaround
When writing to the Pause Register, Switch should be used in the instruction, to ensure that pause register is written before trying to execute the next instruction.	

Programming Note	
Context:	Pause Register
When writing to the Pause Register, Switch should be used in the instruction, to ensure that pause register is written before trying to execute the next instruction.	

Register and Subregister Numbers for Performance Register

RegNum[3:0]	SubRegNum[4:0]
0000b = <i>tm0</i>	00000b = tm0.0:ud
All other encodings are reserved.	00100b = tm0.1:ud
	01000b = tm0.2:ud
	01100b = tm0.3:ud
	10000b = tm0.4:ud



RegNum[3:0]	SubRegNum[4:0]
	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
4 tm0.4:ud (tp0)	31:16	Reserved
	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

Attribute	Value
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 undeterministic 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.

Description
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.

Description
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.

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.

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.

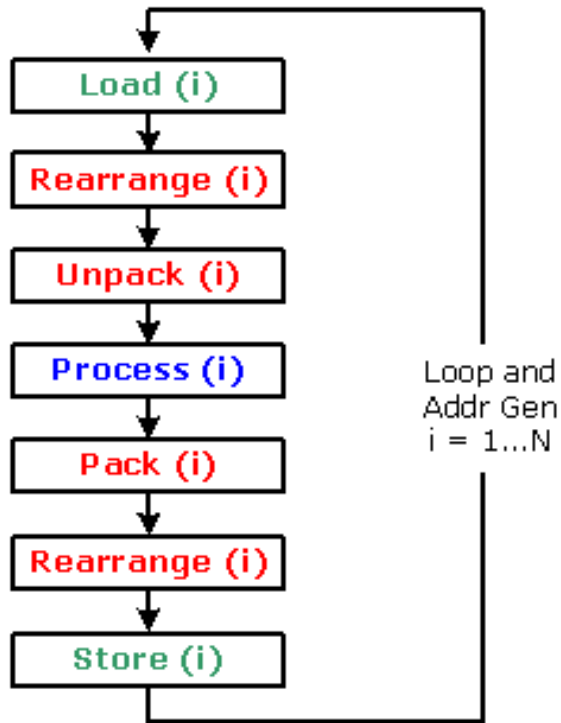
When an immediate vector is used in an instruction, the destination must be 128-bit aligned with destination horizontal stride equivalent to a word for an immediate integer vector (**v**) and equivalent to a DWord for an immediate float vector (**vf**).

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.

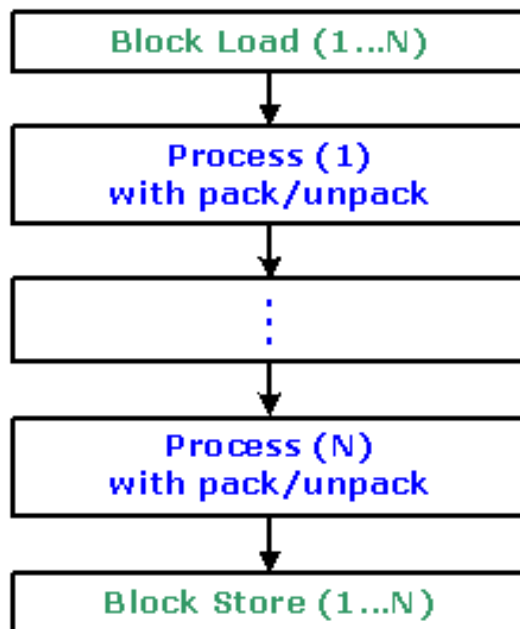
This can be illustrated by the example shown in Conventional SIMD Instruction Sequence and SIMD Instruction Sequence for the Same Program. As shown in Conventional SIMD Instruction Sequence, a sequence of SIMD instruction is executed on a conventional load/store-based superscalar machine with SIMD instruction 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



B.6899-01

SIMD Instruction Sequence for the Same Program



B.6900-01

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.

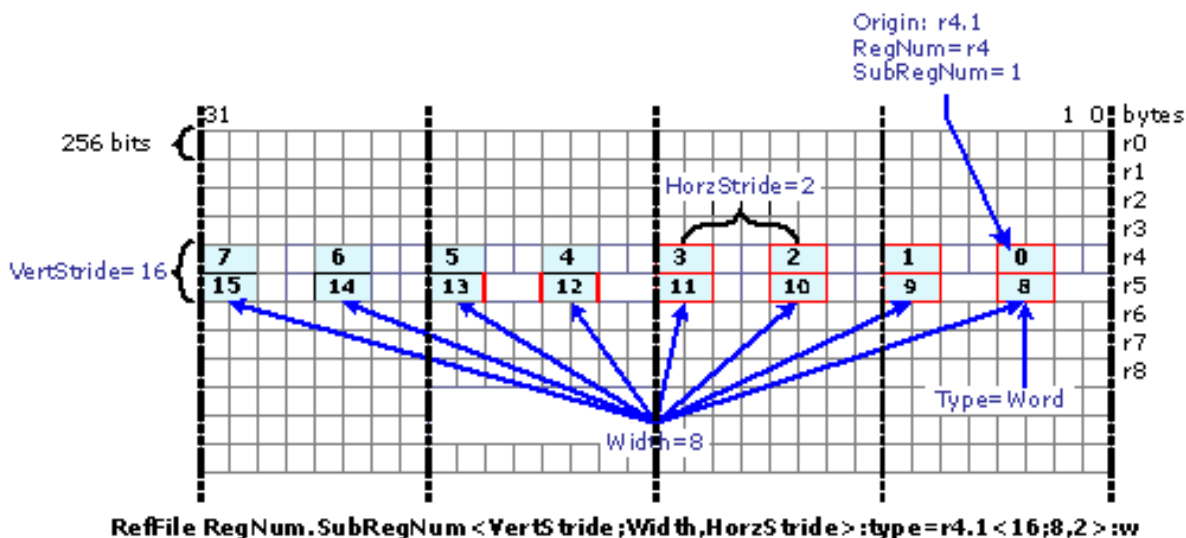
- For an instruction with two source operands, the sizes of the two source operands must be the same.
- The size of a destination operand generally matches the execution size, therefore equals to the number of source operand(s) in the same instruction.
 - Exception of this rule is present for the integer 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



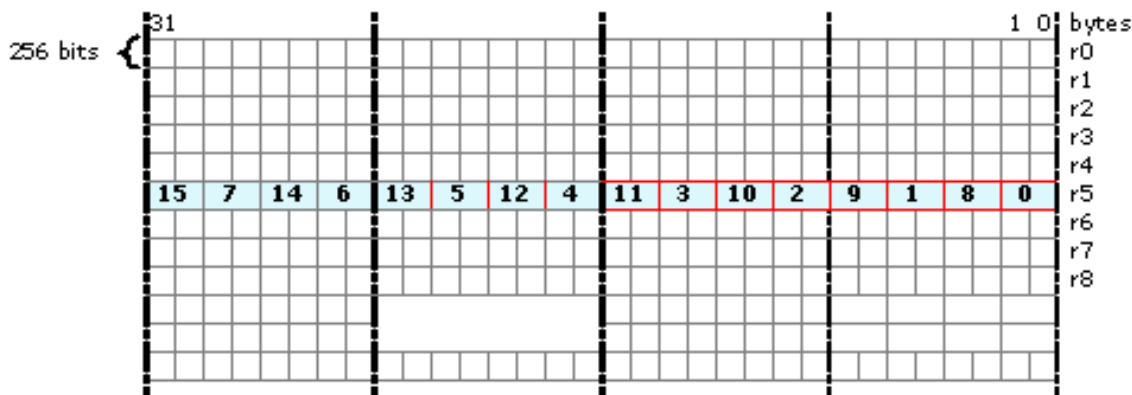
B6901-01

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 ($r5.0<1;8,2>:w$)



RefFile RegNum.SubRegNum <VertStride;Width,HorzStride>:type=r5.0<1;8,2>:w

B 6902-01

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;
}
```

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.

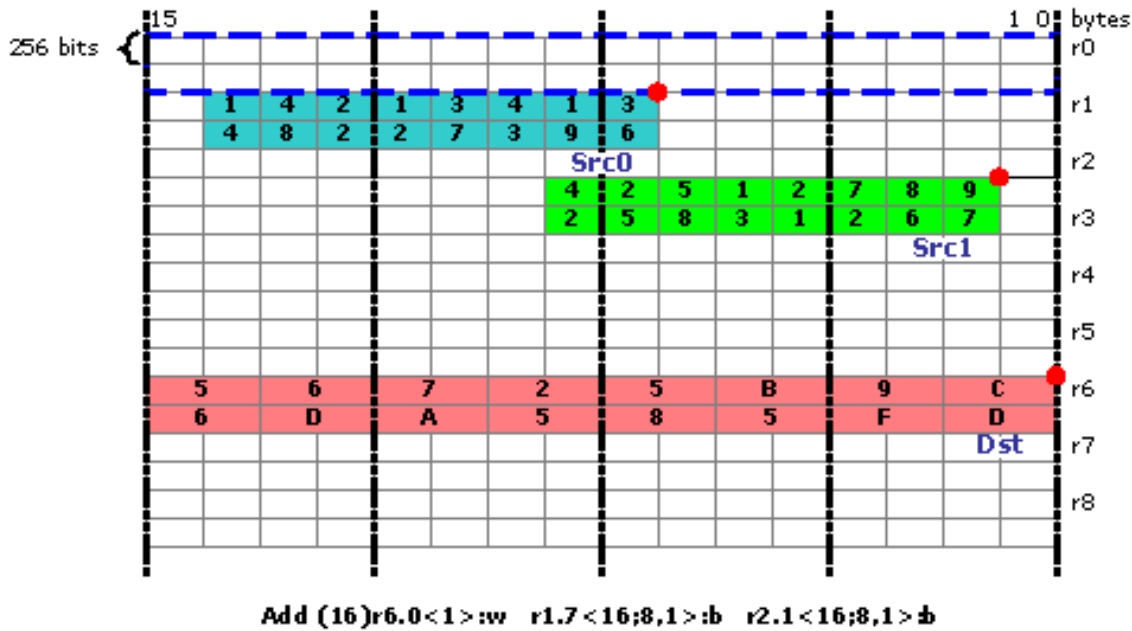
Direct Register Addressing

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 $C = 3+9$.

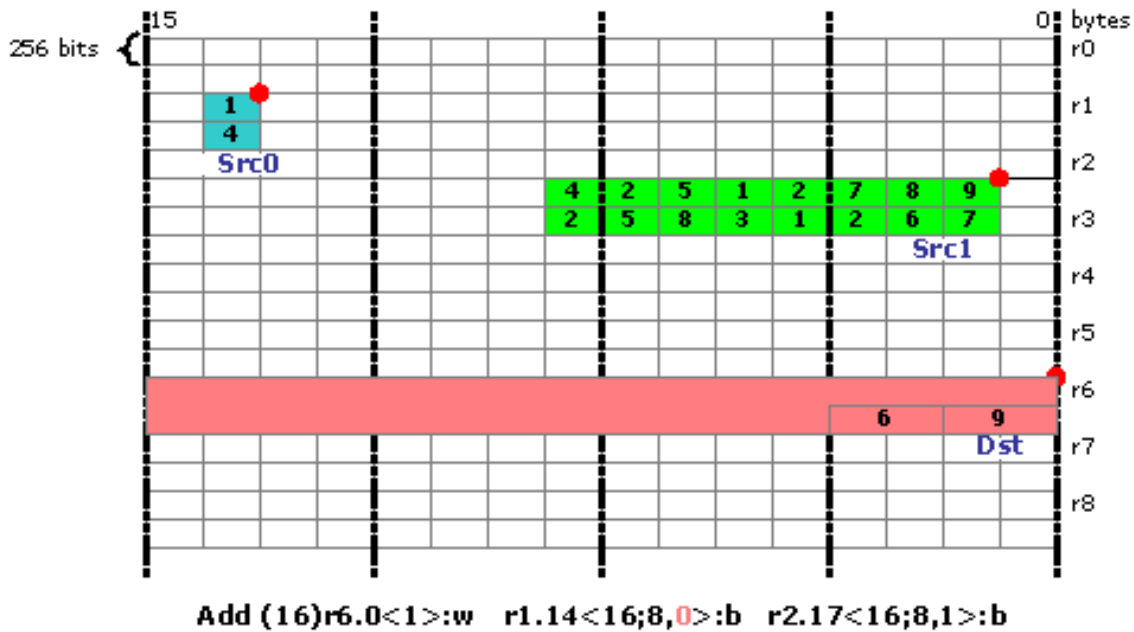
A region description example in direct register addressing



B6903-01

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



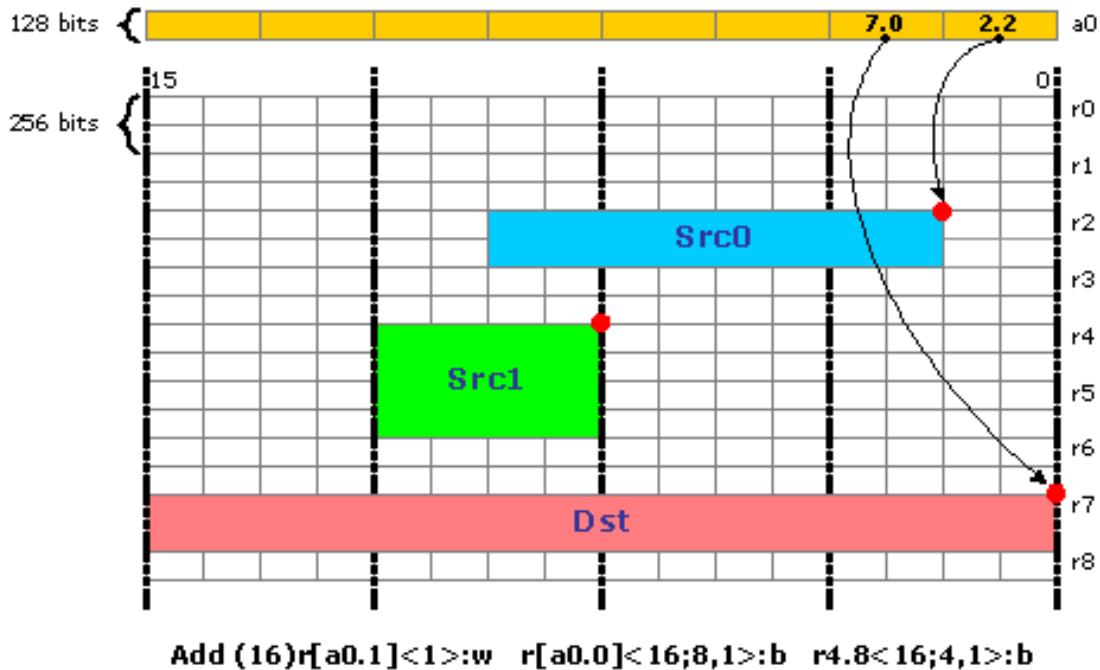
B6904-01

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



B.6905-01

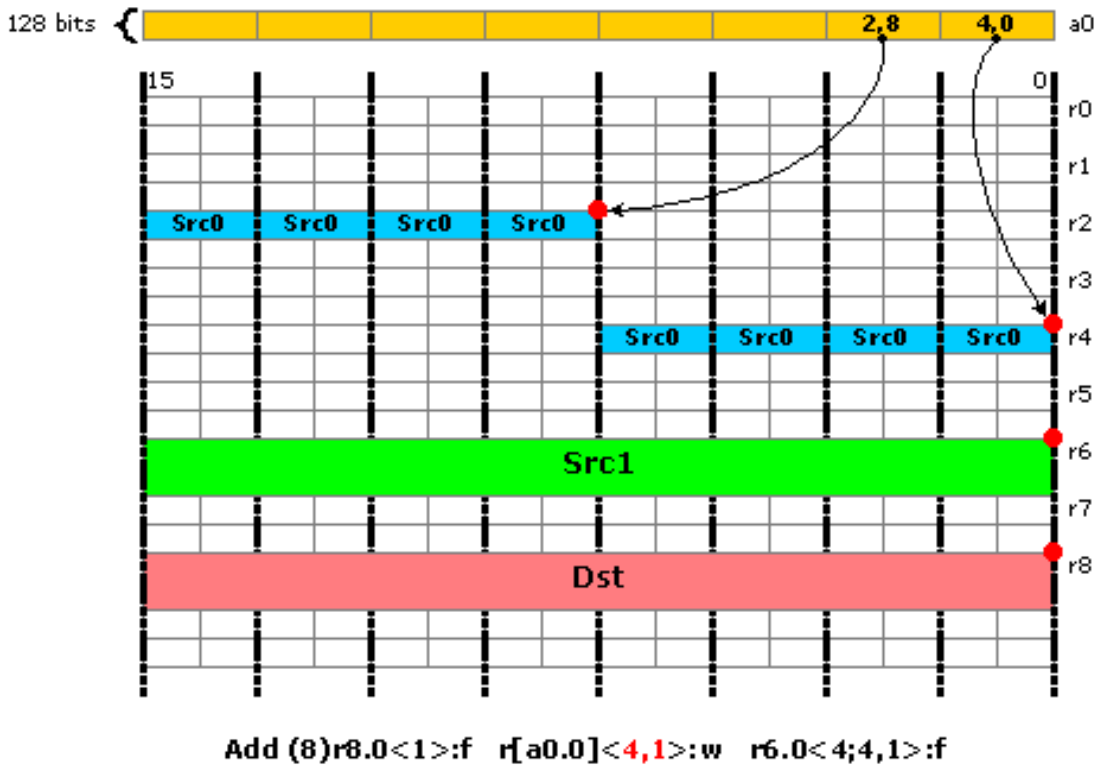
Register-Indirect Register Addressing with a Vx1 Index Region

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*.

Within the 16-bit address register, bits 15:5 determine RegNum and bits 4:0 determine SubRegNum.

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)



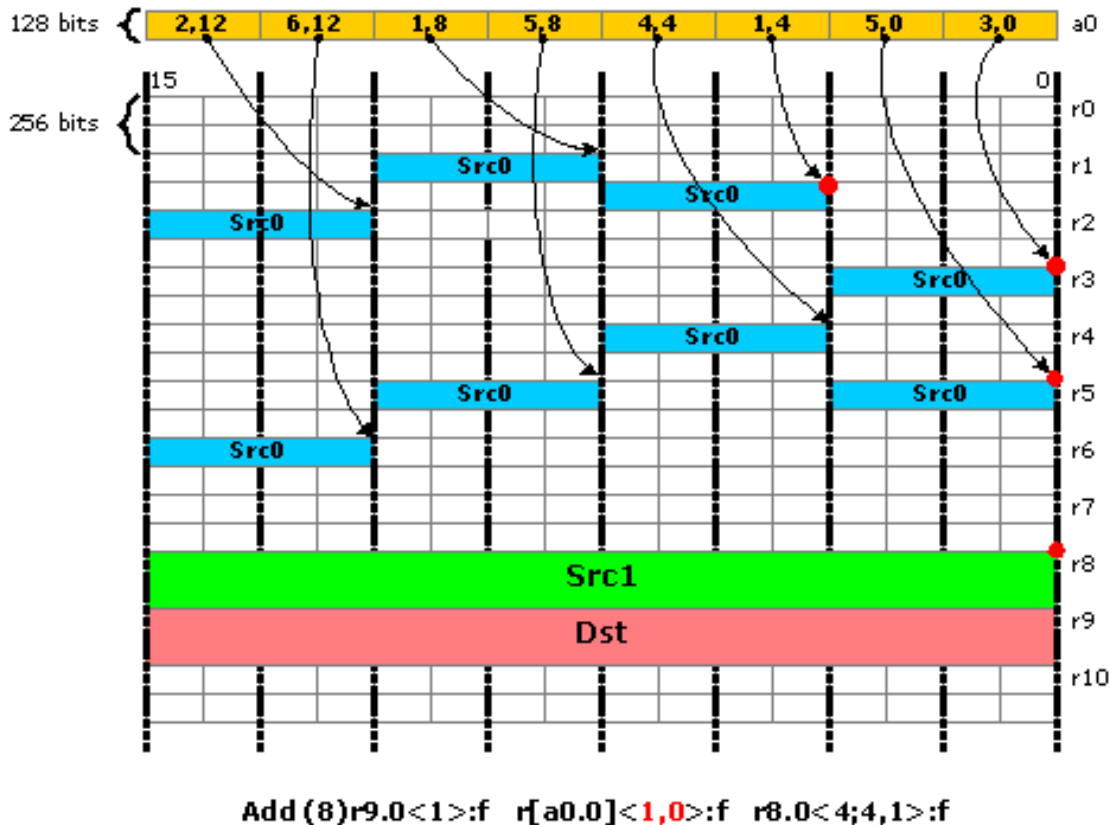
B.6906-01

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).



B6907-01

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-byte 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:

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 $HorzStride \neq 0$, *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. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

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.
```

- 3.

5. Special Cases for Word Operations

There are some special cases for word 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 not show and there are no special cases in this category.

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 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.

Special Requirements for Handling 64bit Precision Data Types

Requirement
In Align16 mode, all regioning parameters must use the syntax of a pair of packed floats,

Requirement
<p>including channel selects and channel enables.</p> <pre>// Example: mov (8) r10.0.xyzw:df r11.0.xyzw:df // The above instruction moves four double floats. The .x picks the // low 32 bits and the .y picks the high 32 bits of the double float.</pre>
<p>In Align1 mode, all regioning parameters like stride, execution size, and width are in units of element size. However in Align16 mode, the channel selects and channel enables must always be used in pairs of packed floats, because these parameters are defined for DWord elements ONLY.</p> <pre>// Example: mov (4) r10.0<1>:df r11.0<4;4,1>:df // The above instruction moves four double floats.</pre>
<p>ARF registers must never be used with 64b datatype or when operation is integer DWord multiply.</p>
<p>When source or destination datatype is 64b or operation is integer DWord multiply, DepCtrl must not be used.</p>

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.

Requirement
<p>In Align16 mode, when half float and float data types are mixed between source operands OR between source and destination operands, the register content are assumed to be packed. In such cases the execution size reflects the number of float elements. Since a stride of 1 is assumed, source is selected in packed form and 16 bit packed data is updated on the destination operand, if the datatype is half-float.</p>
<p>For Align16 mixed mode, both input and output packed f16 data must be oword aligned, no oword crossing in packed f16.</p> <p>Examples:</p> <pre>Case (a) mad (8) r10.0.xy:hf r11.0.xxxx:f r12.xyzw:hf r13.yyyy:hf // The 16b of each word (r12.0, r12.1, r12.2, r12.3.. and so on) forms the source operand. // r13.1 and r13.5 is replicated for source operand. // The lower 16b of a Dword is updated for destination. With channel enables .xy , r10.0, r10.1, r10.4 and r10.5 are updated.</pre> <pre>Case (b) mad (8) r10.0.xy:f r11.0.xxxx:f r12.xyzw:hf r13.yyyy:hf // The example is similar to Case(a), except that entire DWord is updated on the destination.</pre>
<p>In Align16 mode, replicate is supported and is coissueable.</p> <pre>mad(8) r20.xyzw:hf r3.0.r:f r6.0.xyzw:hf r6.0.xyzw:hf {Q1}</pre>

Requirement
<p>No SIMD16 in mixed mode when destination is packed f16 for both Align1 and Align16.</p> <pre>mad(8) r3.xyzw:hf r4.xyzw:f r6.xyzw:hf r7.xyzw:hf add(8) r20.0<1>:hf r3<8;8,1>:f r6.0<8;8,1>:hf {Q1}</pre>
<p>No accumulator read access for Align16 mixed float.</p>
<p>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.</p>
<p>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.</p> <pre>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</pre>
<p>In Align16, vertical stride can never be zero for f16</p> <pre>add(8) r3.xyzw:hf r4.0<4>xyzw:f r6.0<0>.xyzw:hf</pre>
<p>Math operations for mixed mode:</p> <ul style="list-style-type: none"> - In Align16, only packed format is supported <pre>math(8) r3.xyzw:hf r4.0.<4>xyzw:f r6.0<0>.xyzw:hf 0x09</pre> <ul style="list-style-type: none"> - In Align1, f16 inputs need to be strided <pre>math(8) r3<1>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf</pre>
<p>In Align1, destination stride can be smaller than execution type. When destination is stride of 1, 16 bit packed data is updated on the destination. However, output packed f16 data must be oword aligned, no oword crossing in packed f16.</p> <pre>add(8) r3<1>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf mov(8) r3<1>:hf r4.0<8;8,1>:f</pre>
<p>For mixed float operations, f16 datatype write to accumulator cannot be packed destination.</p>
<p>Mixed mode instruction allows fp16 operands in the following cases:</p> <ul style="list-style-type: none"> • dst, src0 and src1 for 2-source instructions not involving multiplier (add, cmp etc). • dst and src0 for 2-source instructions involving multiplier (mul, mac etc). • dst, src0 and src1 for 3-source instructions (mad). <p>Mixed mode instruction allows packed fp16 destination.</p>

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.

Rules

1. 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
```

2. 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.
```

3. Indirect addressing on src1 must be a 1x1 indexed region mode.
4. When a Vx1 or 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.
```

Rules
5. Indirect addressing on the destination must be a 1x1 indexed region mode.
Execution size of 32 is NOT supported in Vx1 or VxH modes.
Accumulator as destination is NOT supported in Vx1 or VxH modes.

1. 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
All flow control (branching) instructions must use the Align1 access mode.
When using Align16 mode for conversion of data elements of different sizes, both source and destination must be one register each.
<p>In Align16 mode, each destination register gets all data from one source register. This means, the data for one destination register is never scattered across two source registers.</p> <pre>// Example: // Allowed - all sources are contained within one register. mul (8) r10.0:f r11.0:f r12.4<0>:f // NOT Allowed - src1 (r14) is scattered across two registers. mad (8) r10.0:f r12.0<0>:f r14.4:f r16.0:f</pre>
<p>Conversion between Integer and HF (Half Float) must be DWord-aligned and strided by a DWord on the destination.</p> <pre>// Example: add (8) r10.<2>:hf r11.<8;8,1>:w r12.<8;8,1>:w // Destination stride must be 2. mov (8) r10.<2>:w r11.<8;8,1>:hf // Destination stride must be 2.</pre>
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.
<p>Regioning Rules for Align1 Ternary Operations</p> <p>Width is an implied regioning parameter.</p> <ol style="list-style-type: none"> Width is 1 when Vertical and Horizontal Strides are both zero (broadcast access). Width is equal to Vertical Stride when Horizontal Stride is zero. Width is equal to Vertical Stride/Horizontal Stride when both Strides are non-zero. Vertical Stride must not be zero if Horizontal Stride is non-zero. This implies Vertical Stride is always greater than Horizontal Stride. 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'. The Source 2 operand must be 64-bit aligned.

Restriction
<p>src2 regioning removal</p> <ol style="list-style-type: none"> 1. Byte datatype is not supported 2. Broadcast is supported (<0>) 3. When source 2 is not a scalar, regioning must follow these rules: <ul style="list-style-type: none"> • Source 2 offset and destination offset must be aligned to the same word e.g. mad(4) e.g. r10.2<1>:hf r20:f r21:hf r22.2<1>:hf • Source 2 and destination strides in bytes must be same. • In mixed mode cases, source2 stride needs to be aligned to the maximum datatype e.g. mad(4) r10.2<1>:f r20:f r21:hf r22.2<2>:hf
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.
<p>In an atomic chain of instructions in a macro the following applies:</p> <ul style="list-style-type: none"> • 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 discusses SIMD execution, both with and without predication. See the subtopics for more details.

Predication

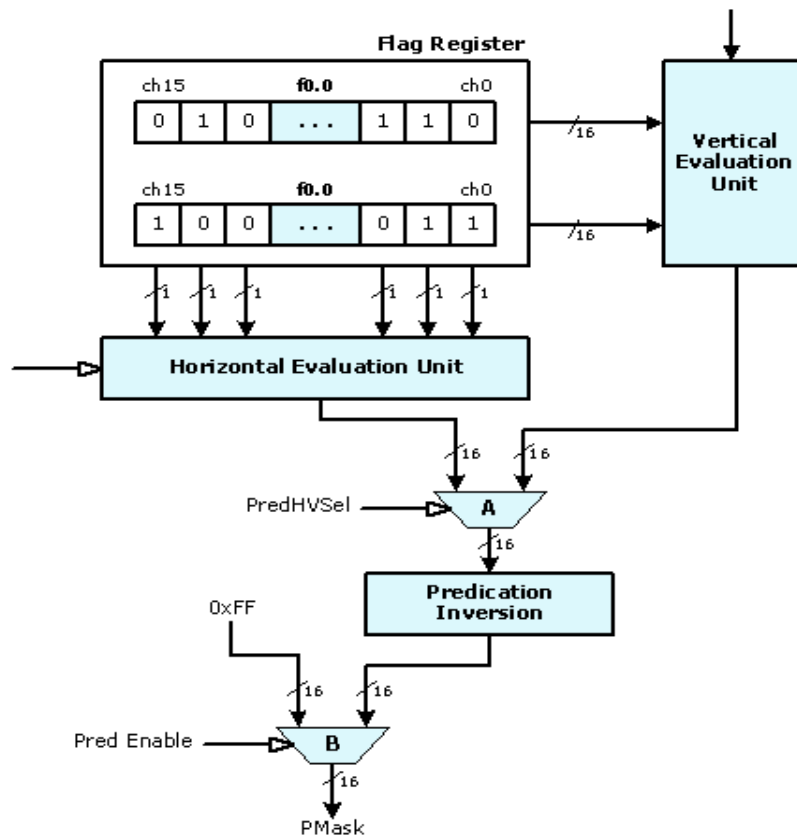
Predication is the conditional SIMD channel selection for execution on a per instruction basis. It is an efficient way of dynamic SIMD channel enabling without paying branch instruction overhead. When

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.

Predications 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 Inversion 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



B.6908-01

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 '.any4h':    groupSize = 4; <op> = '|'; break;
        Case '.any8h':    groupSize = 8; <op> = '|'; break;
        Case '.any16h':   groupSize = 16; <op> = '|'; break;
        Case '.all2h':    groupSize = 2; <op> = '&'; break;
        Case '.all4h':    groupSize = 4; <op> = '&'; break;
        Case '.all8h':    groupSize = 8; <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 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: 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.

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.
- **OF** bit: This bit reflects whether an overflow occurred 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.
- **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 that the bits generated at the output of a compute are before the **.sat**.

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 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. All other internal conditional signals are ignored.
.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.

Description
<p>Instructions other than <code>send</code>, may use this control as long as operations that have different pipeline latencies are not mixed. The operations that have longer latencies are:</p> <ul style="list-style-type: none"> ▪ Opcodes <code>pln, lrp, dp*</code>. ▪ 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 filled 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 instructions (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 FPU's 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 *rm.n<HorzStride>:type*. General register directly addressed source regions use the syntax *rm.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. Supports up to 32 channels.
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 as their values approach zero, called <i>gradual underflow</i> .
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 <i>send</i> or <i>sendc</i> 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 <i>channels</i> . 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

Term	Description
	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, <i>instruction</i> 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 instruction	A 128-bit instruction, the regular instruction format that allows all defined instruction parameters 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	<p>A register region is described as <i>packed</i> 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.</p> <p>The immediate vector data types are all described as <i>Packed</i> because each such type packs several small data elements into a 32-bit immediate value.</p>
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.

Term	Description
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, PclP[n] and each channel n is only eligible to execute, subject to other masking, when PclP[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 <i>send</i> or <i>sendc</i> 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. <i>Word</i> 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	4	32	I	Unsigned integer in the range 0 to $2^{32} - 1$.

Short Name	Assembler Syntax	Long Name	Size in Bytes	Size in Bits	Integral or Float	Description
		Integer				
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 <i>mac</i> (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.

Name	Assembler Syntax	Description
Flags	fr.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 <i>mov</i> 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 EUs 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 <i>Message Gateway, Media</i>)
synchronization of thread output to memory buffers (see <i>Geometry Shader</i>)

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

This section describes how the EU changes. 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
<p>These features or behaviors are added:</p> <ul style="list-style-type: none"> • The maximum <i>ExecSize</i> increases to 32, for byte or word operands. • Increase the number of flag registers from one to two. • Add the <i>NibCtrl</i> field, used with <i>QtrCtrl</i> 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. • Add bit manipulation instructions: <i>bfi1</i>, <i>bfi2</i>, <i>bfrev</i>, <i>cbit</i>, <i>fbh</i>, and <i>fbl</i>. • Add the integer <i>addc</i> (Add with Carry) and <i>subb</i> (Subtract with Borrow) instructions. • Add the <i>brc</i> (Branch Converging) and <i>brd</i> (Branch Diverging) instructions. • For the <i>cmp</i> and <i>cmpn</i> instructions, relax the accumulator restrictions. • For the <i>sel</i> instruction, remove the accumulator restriction. • Add the Rounding Mode and Double Precision Denorm Mode fields in Control Register 0. • 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: <i>brc</i>, <i>brd</i>, <i>call</i>, and <i>jmpj</i>. • A <i>call</i> instruction can get the relative instruction offset (JIP) from a register. • Add the <i>calla</i> (Call Absolute) instruction. • A <i>mov</i> instruction with different source and destination types can now use conditional modifiers. • 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., <i>avg</i>) 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.

Description

- 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.
- 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.
- 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.
- 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

Description
broadcast. <ul style="list-style-type: none"> • Add 8bits of software hints in instruction to facilitate scheduling. • Byte datatype support removed from source1.

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.

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

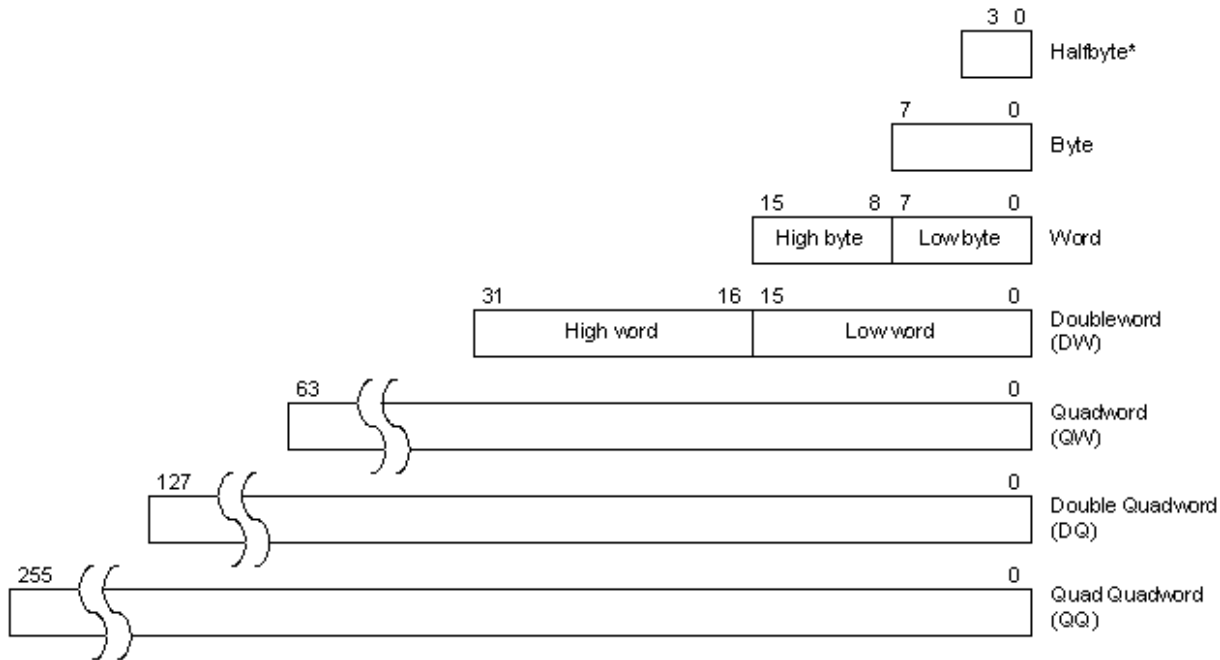
Floating Point Modes

- 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.

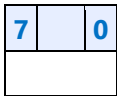
Numeric Data Types

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.

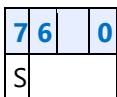
Integer Numeric Data Types

The Execution Unit supports the following integer data types. Signed integer types use two's complement representation for negative numbers.

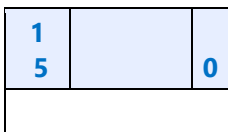
UB: Unsigned Byte, 8-bit Unsigned Integer



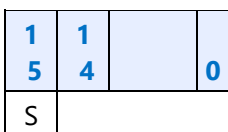
B: Byte, 8-bit Signed Integer



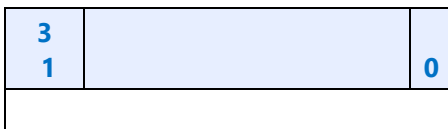
UW: Unsigned Word, 16-bit Unsigned Integer



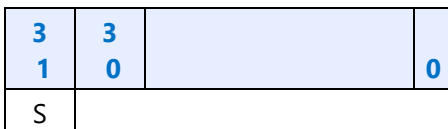
W: Word, 16-bit Signed Integer



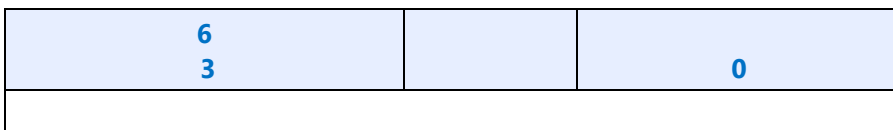
UD: Unsigned Doubleword, 32-bit Unsigned Integer



D: Doubleword, 32-bit Signed Integer



UQ: Unsigned Quadword, 64-bit Unsigned Integer





Q: Quadword, 64-bit Signed Integer

6 3	6 2		0
S			

UV: Packed Unsigned Half-Byte Integer Vector, 8 x 4-Bit Unsigned Integer

3 1		2 8	2 7		2 4	2 3		2 0	1 9		1 6	1 5		1 2	1 1		8	7		4	3		0

V: Packed Signed Half-Byte Integer Vector, 8 x 4-Bit Signed Integer

3 1		2 8	2 7		2 4	2 3		2 0	1 9		1 6	1 5		1 2	1 1		8	7		4	3		0
S		S		S		S		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.
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.

HF: Half Float, 16-bit Half-Precision Floating-Point Number

1 5	1 4		1 0	9		0
S	biased exp.			fraction		

F: Float, 32-bit Single-Precision Floating-Point Number

3 1	3 0		2 3	2 2		0
S	biased exponent			fraction		

DF: Double Float, 64-bit Double-Precision Floating-Point Number

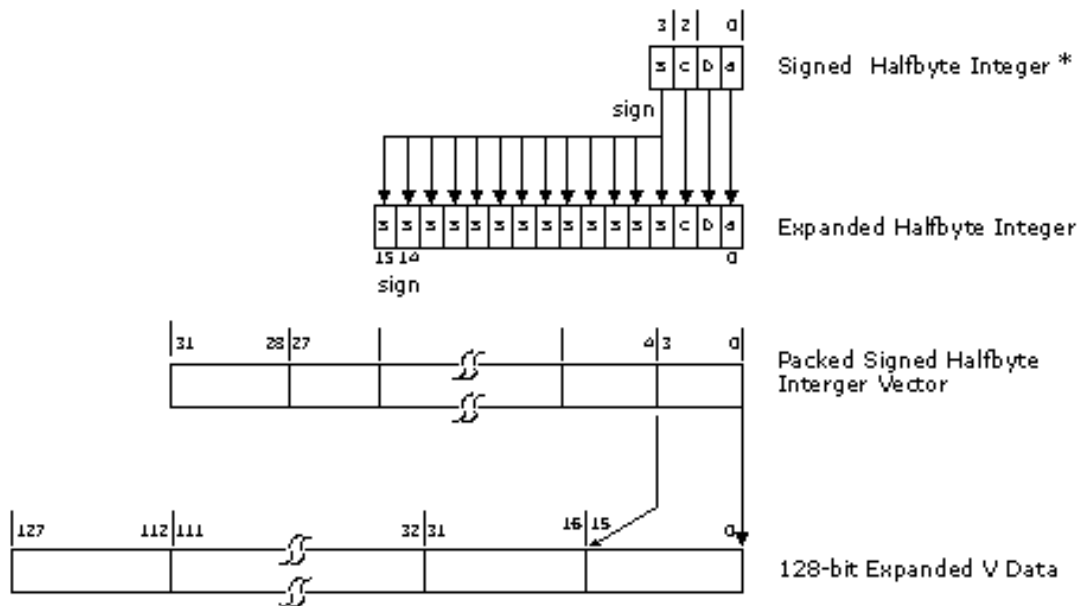
6 3	6 2		5 2	5 1		0
S	biased exponent			fraction		

Packed Signed Half-Byte Integer Vector

A packed signed halfbyte integer vector consists of 8 signed halfbyte integers contained in a doubleword. Each signed halfbyte 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 short hand format notation for a packed signed half-byte vector is **V**.

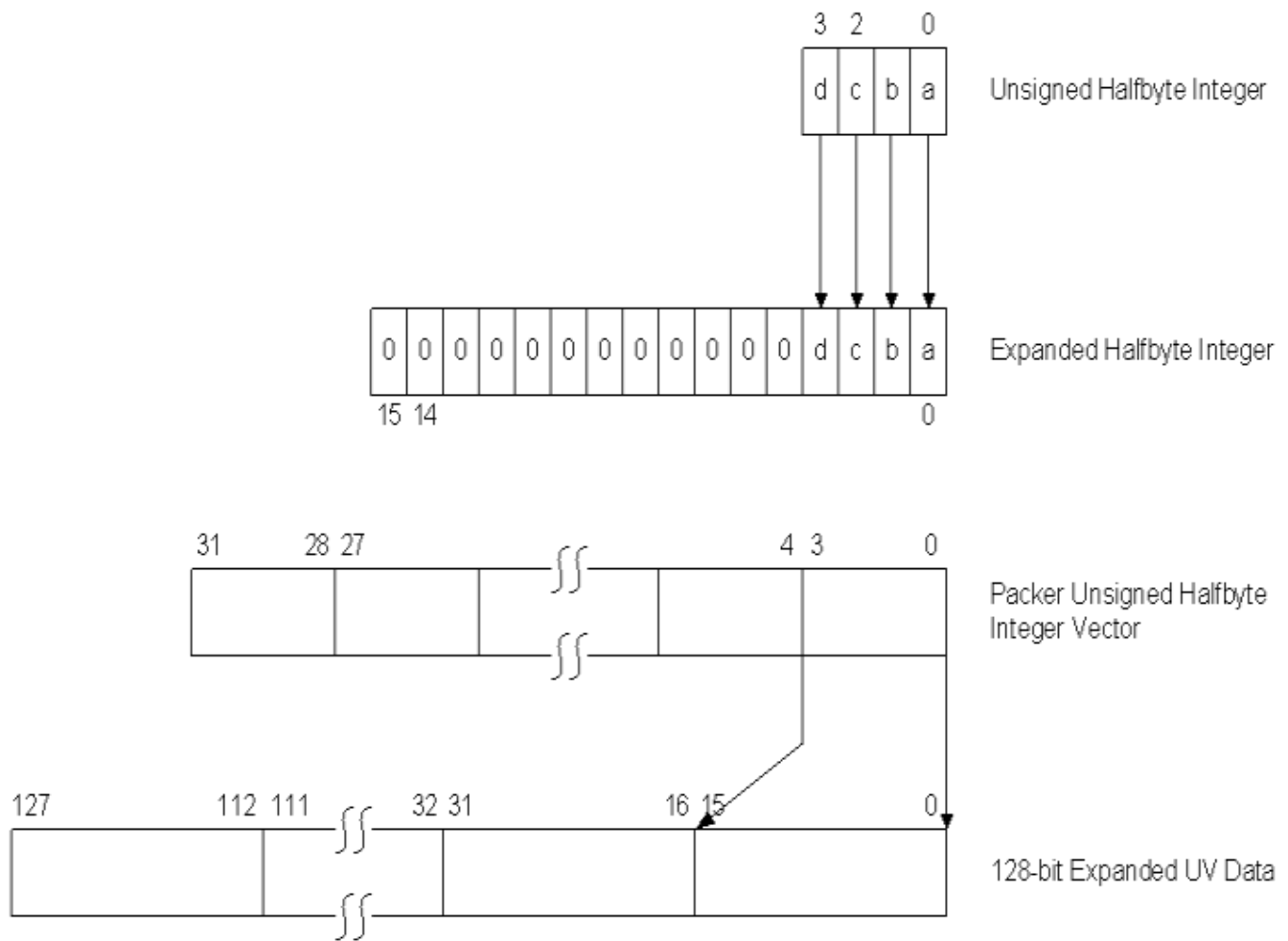
Converting a Packed Half-Byte Vector to a 128-bit Signed Integer Vector



B.6885-01

Packed UnSigned Half-Byte Integer Vector

A packed unsigned halfbyte integer vector consists of 8 unsigned halfbyte integers contained in a doubleword. Each unsigned halfbyte 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.

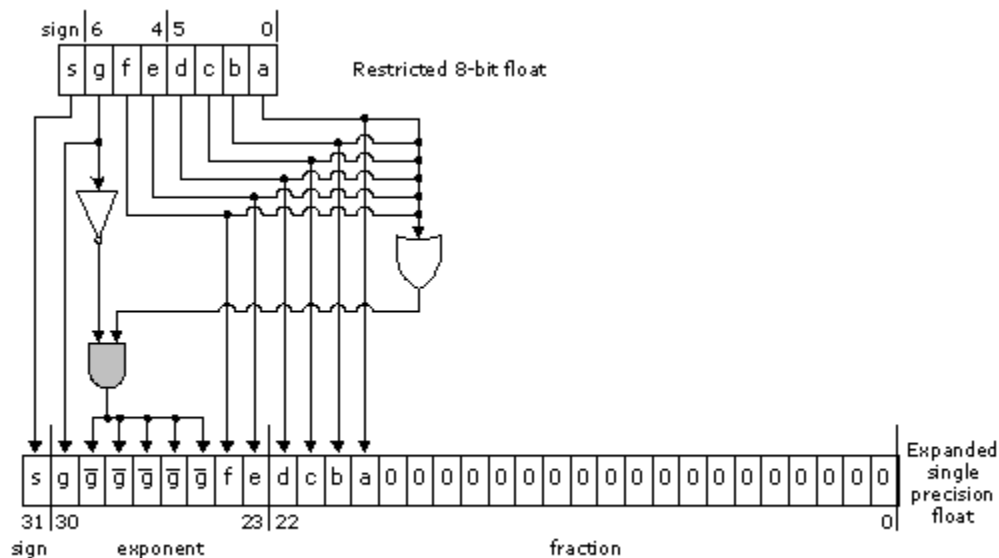


Packed Restricted Float Vector

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



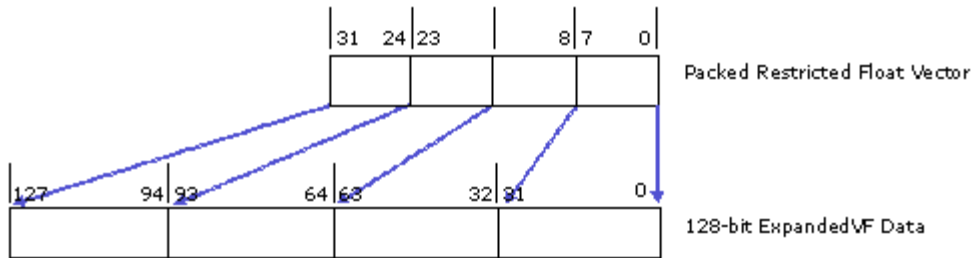
The following table shows all possible numbers of the restricted 8-bit float. Only normalized float numbers can be represented, including positive and negative zero, and positive and negative finite numbers. Normalized infinities, NaN, and denormalized float numbers cannot be represented by this type. 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.



B6889-01

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 details 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: <ul style="list-style-type: none"> • 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. • Mixed mode operations should always behave like sum of individual operations.

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, 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 computation result. 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.

Note that this mode is applicable ONLY to Single Precision Float datatype.

Description
This also implies that ALT mode is not supported when Single precision datatype is involved in format conversion to double precision or half precision.
ALT_MODE is supported for Single Precision float ONLY. Hence, ALT_MODE is NOT supported in mixed mode operation.

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"
(b) Sampler honors 0/0 = 0 as if (1/0)*0	There is no divide in Sampler
l 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 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.

Compared to previous generations, does the following:

- 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 input 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 $\leq 0.5 \text{ ulp}$ (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 arbitrary 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 $\geq 8,388,608$ (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, *lmin* is the smallest representable value in a signed integer type, *lmax* 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	lmax	Invalid Operation
$f > lmax$	lmax	Invalid Operation
$lmin \leq f \leq lmax$	f	Inexact if rounding changed f
$f < lmin$	lmin	Invalid Operation
-inf	lmin	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	lmax	Invalid Operation
$f > lmax$	lmax	Invalid Operation

FP Value	Integer Result	FP Exception
$0 \leq f \leq I_{max}$	f	Inexact if rounding changed f
$f = -0$	0	Inexact
$-1 < f < -0$	0	Inexact
$-1 < f < -0$	Real Indefinite	Invalid Operation
$-f_{max} \leq f \leq -1$	Real Indefinite	Invalid Operation
$-\infty$	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.

$\text{Min}(x, \text{qNaN}) = \text{Min}(\text{qNaN}, x) = x$ with no exceptions signaled.

$\text{Min}(x, \text{sNaN}) = \text{Min}(\text{sNaN}, x) = \text{qNaN}$ (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.

$\text{Min}(x, \text{sNaN}) = \text{Min}(\text{sNaN}, x) = x$

$\text{Max}(x, \text{qNaN}) = \text{Max}(\text{qNaN}, x) = x$ with no exceptions signaled.

$\text{Max}(x, \text{sNaN}) = \text{Max}(\text{sNaN}, x) = \text{qNaN}$ (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.

$\text{Max}(x, \text{sNaN}) = \text{Max}(\text{sNaN}, x) = x$

Special cases(when both sources are NaN inputs)



$\text{Min}(qNaN, qNaN) = qNaN$ (of the first source) and no exception raised

$\text{Min}(qNaN, sNaN) = \text{Min}(sNaN, qNaN) = qNaN$ (quiet-ized sNaN and signal the Invalid Operation exception)

$\text{Min}(sNaN, sNaN) = qNaN$ (of the first source and signal Invalid Operation Exception)

$\text{Max}(qNaN, qNaN) = qNaN$ (of the first source) and no exception raised

$\text{Max}(qNaN, sNaN) = \text{Max}(sNaN, qNaN) = qNaN$ (quiet-ized sNaN and signal the Invalid Operation exception)

$\text{Max}(sNaN, sNaN) = qNaN$ (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 per thread, shared by all channels. (Maintaining separate per channel exception flags for 32 channels would require 160 bits per thread.)

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:

$+\text{inf} - (+\text{inf})$ or $(-\text{inf}) - (-\text{inf})$

$\pm 0 / \pm 0$

$\pm \text{inf} / \pm \text{inf}$

$\pm 0 \times \pm \text{inf}$ or $\pm \text{inf} \times \pm 0$

Remainder($\pm x, \pm 0$)

Remainder($\pm \text{inf}, \pm x$)

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 $\pm x / \pm 0$, where x is a positive, finite, nonzero, and normalized number, signals the Division by Zero Exception and produces a correctly signed $\pm \text{inf}$ result. Note that in accordance with the standard, $\pm 0 / \pm 0$ 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 $\text{LOG}_2(\pm 0)$ signals the Division by Zero Exception and produces $-\text{inf}$ 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 $> f_{\text{max}}$, 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 $\pm \text{inf}$ or $\pm f_{\text{max}}$ 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 ($\text{abs}(\text{rounded unbounded result}) > f_{\text{max}}(\text{destination format})$) {

Set the Overflow Exception flag to 1.

Output = $\pm \text{inf}$ or $\pm f_{\text{max}}$ 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 $\text{abs}(x) < f_{\text{min}}$, where $f_{\text{min}} = 2^{E_{\text{min}}}$. See the Floating-Point Type Parameters table for the f_{min} and E_{min} 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 f_{min} but nonzero AND the denorm result in the destination format is inexact, then signal the Underflow Exception.

The rounded result can be +/- 0, +/- f_{min} , 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 ($0 < \text{abs}(\text{rounded unbounded result}) < f_{\text{min}}(\text{destination format})$) {
 - if (flush denorms to zero) {
 - Output = 0; // No underflow exception.

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	+denor m	+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
+denor m	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	+denor m	+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
+denor m	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.								
^	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	+denor m	+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
+denor m	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	+denor m	+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
+denor m	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	+denor m	+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
+denor m	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	+denor m	+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
+denor m	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	+denor m	+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
+denor m	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 or -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 (UD, D, UW, W, UB and B).

Input Format	Output Format					
	UD	D	UW	W	UB	B
+/- Zero	00000000	00000000	00000000	00000000	00000000	00000000
+/- Denorm	00000000	00000000	00000000	00000000	00000000	00000000
NAN	00000000	00000000	00000000	00000000	00000000	00000000
-NAN	00000000	00000000	00000000	00000000	00000000	00000000
INF	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-INF	00000000	80000000	00000000	00008000	00000000	00000080
+2 ³² (*)	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-2 ³² -1 (*)	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	<ul style="list-style-type: none"> 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 enable 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.

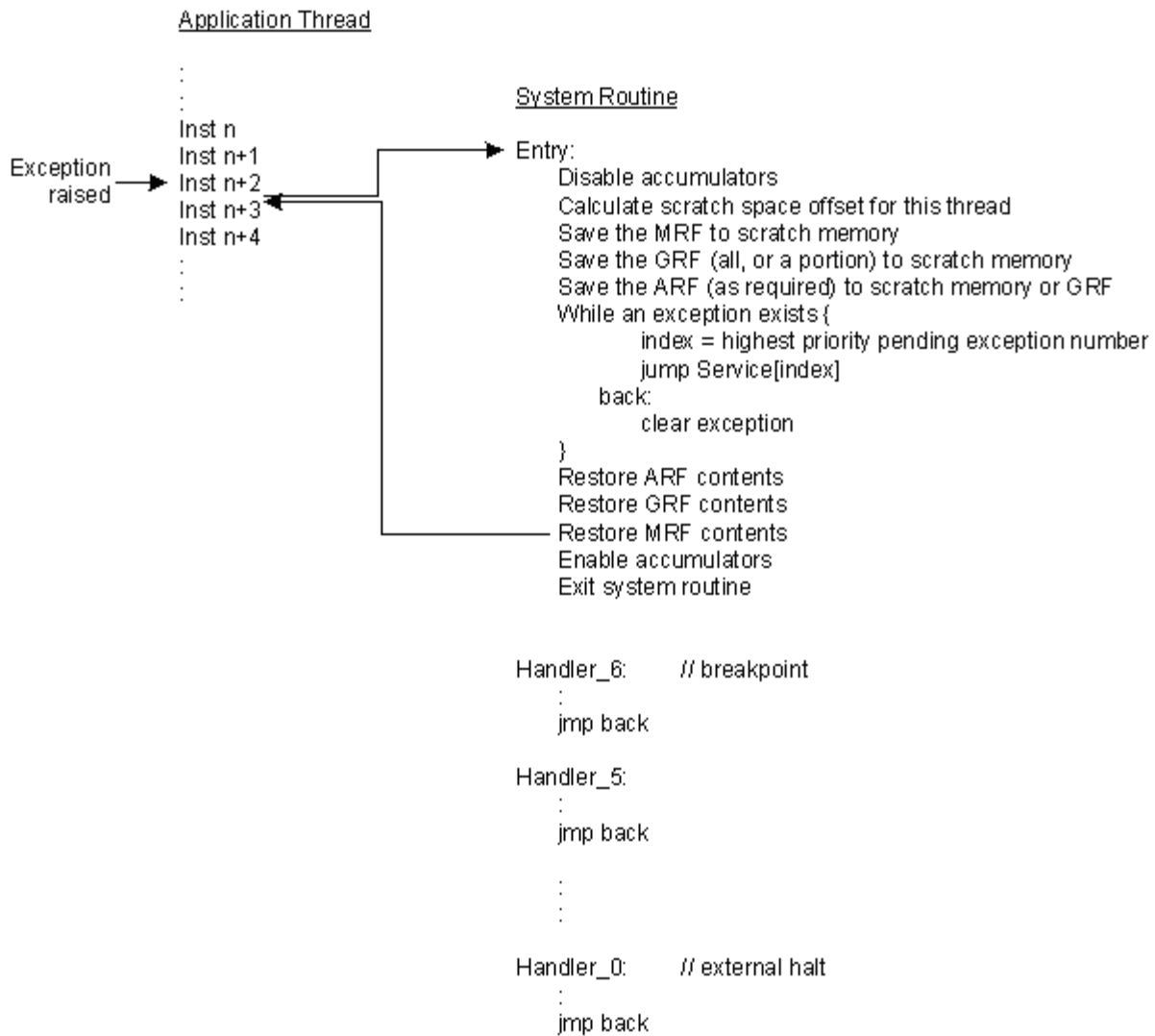
Exception-Related Architecture Registers

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).

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.

$$\text{Per_Thread_Block_Size} = \text{System_Scratch_Block_Size} / (\text{EU_Count} * \text{Thread_Per_EU});$$

$$\text{Offset} = (\text{sr0.0.EID} * \text{Threads_Per_EU} + \text{sr0.0.TID}) * \text{Per_Thread_Block_Size};$$

where in GFX:

$$\text{Threads_Per_EU} = 4$$

$$\text{EU_Count} = 8$$

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 work loads, 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 0xFFFFFFFF
#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:
jmp (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 an 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

Bit	2src	2src-imm	3src
63	Src1.Reg	ImmValue2[11:0]	Src1.Reg
62			
61			
60			
59			
58			
57			
56			
55	Src1.Index2		Src2.Reg
54			
53			
52			
51	Src0.Index2	Src0.Index2	
50			
49			

Bit	2src	2src-imm	3src
48			
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	ControllIndex2	ControllIndex2	ControllIndex3
27			
26			
25			
24			
23	Dst.Reg	Dst.Reg	Dst.Reg
22			
21			
20			
19			
18			
17			

Bit	2src	2src-imm	3src
16			
15	SWSB	SWSB	SWSB
14			
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 `ControllIndex2`, `DataTypeId2`, `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.

ControllIndex2 Compact Instruction Field Mappings

ControllIndex2	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)
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)

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
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

Data Type Index 2	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 }
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
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

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 }
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
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

SubRegIndex2	15-Bit Mapping { Src1.SubRegNum[4:0], Src0.SubRegNum[4:0], Dst.SubRegNum[4:0] }	Mapped Meaning { dst, src0, src1 }
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 Compact Instruction Field Mappings

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	0100,011,0,01,00	r<8;8,1>
1	0000,000,0,00,00	r<0;1,0>
2	0100,011,0,01,10	-r<8;8,1>
3	0100,011,0,01,01	(abs)r<8;8,1>
4	0000,000,0,00,10	-r<0;1,0>
5	0010,000,0,00,00	r<2;1,0>
6	0010,010,0,00,00	r<2;4,0>
7	0011,010,0,00,00	r<4;4,0>
8	0010,001,0,01,00	r<2;2,1>
9	0011,000,0,00,00	r<4;1,0>
10	0010,001,0,01,10	-r<2;2,1>
11	0011,010,0,01,00	r<4;4,1>
12	0100,011,0,01,11	-(abs)r<8;8,1>
13	0001,000,0,00,00	r<1;1,0>
14	0000,000,0,00,01	(abs)r<0;1,0>
15	1111,000,1,00,00	r[a]<1,0>

Src1Index2 Compact Instruction Field Mappings

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	00,0100,011,0,01	r<8;8,1>
1	00,0000,000,0,00	r<0;1,0>
2	10,0100,011,0,01	-r<8;8,1>
3	10,0000,000,0,00	-r<0;1,0>
4	01,0100,011,0,01	(abs)r<8;8,1>
5	10,0011,010,0,00	-r<4;4,0>
6	00,0010,000,0,00	r<2;1,0>
7	00,0010,001,0,01	r<2;2,1>
8	10,0010,001,0,01	-r<2;2,1>
9	00,0011,010,0,00	r<4;4,0>
10	00,0011,010,0,01	r<4;4,1>
11	00,0011,000,0,00	r<4;1,0>
12	11,0100,011,0,01	-(abs)r<8;8,1>
13	01,0000,000,0,00	(abs)r<0;1,0>
14	11,0000,000,0,00	-(abs)r<0;1,0>
15	10,0011,010,0,01	-r<4;4,1>

ImmValue2 Compact Instruction Field

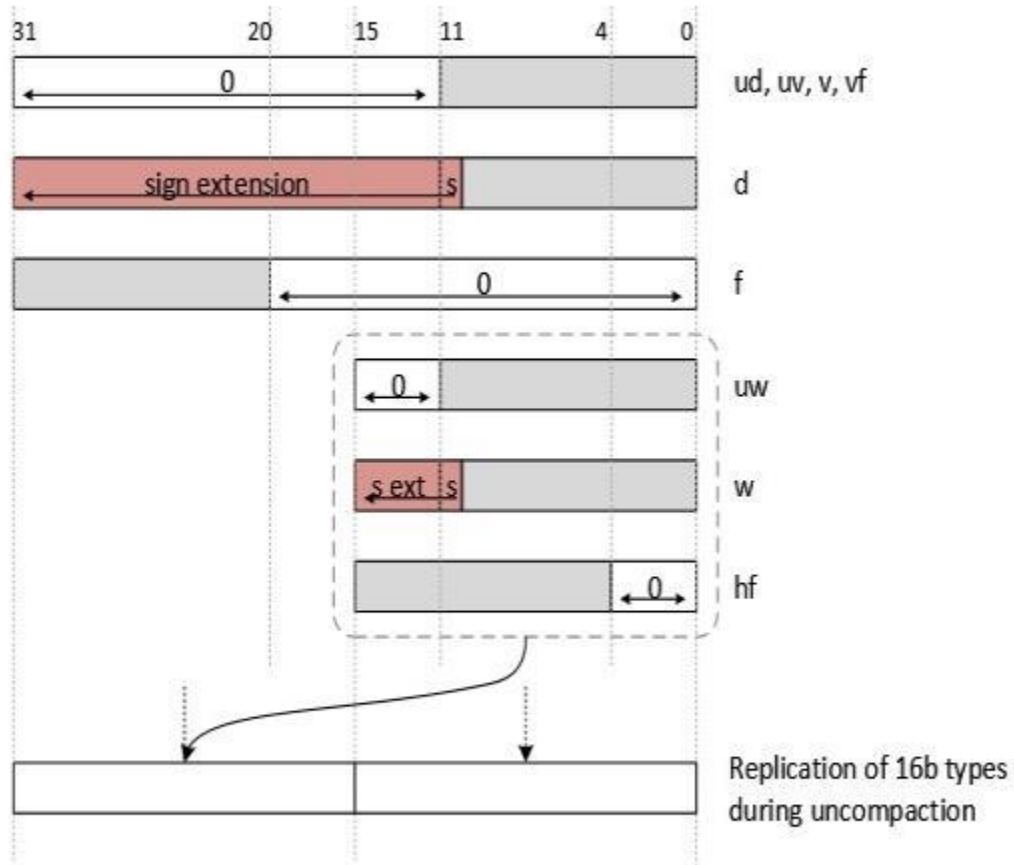
Immediates values of up to 12 bits can be represented in ImmValue2 field of compacted format.

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 uncompactation zero is extended up to the MSB of datatype. 16bit datatypes are replicated to occupy 32bit of uncompactated 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 uncompactation sign bit 11 is extended up to the MSB of datatype. 16bit datatypes are replicated to occupy 32bit of uncompactated 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 uncompactation bits [19:0] ([3:0] for 16d type) are set to zero. 16bit datatypes are replicated to occupy 32bit of uncompactated immediate value.

64bit datatypes are not compacted.



Ternary Instruction Compression Tables

The following three tables describe the mappings for the ControlIndex3, SourceIndex3 and SubRegIndex3 fields in the Ternary Compact Instruction format.

Type or Description
The following three tables describe the mappings for the ControlIndex3, SourceIndex3 and SubRegIndex3 fields in the Ternary Compact Instruction format.



ControllIndex3 Compact Instruction Field Mappings

ControllIndex3	36-Bit Mapping { CondMod[3:0], Src1.SrcType[2:0], Src2.SrcType[2:0], Dst.RegFile[0], Dst.HorzStride[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	0000,010,010,1,0,010,1,010, 0,0,0,0,0,0000,0,0,000,100	(16 M0) grf<1>:f :f :f :f
1	0000,010,010,1,0,010,1,010, 0,0,0,0,0,0000,0,0,000,011	(8 M0) grf<1>:f :f :f :f
2	0000,010,010,0,0,010,1,010, 0,0,0,0,0,0000,0,0,000,011	(8 M0) arf<1>:f :f :f :f
3	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,000,011	(W) (8 M0) grf<1>:f :f :f :f
4	0000,010,010,0,0,010,1,010, 0,0,0,1,0,0000,0,0,000,011	(W) (8 M0) arf<1>:f :f :f :f
5	0000,010,010,0,0,010,1,010, 0,0,0,0,0,0000,0,0,010,011	(8 M8) arf<1>:f :f :f :f
6	0000,010,010,1,0,010,1,010, 0,0,0,0,0,0000,0,0,010,011	(8 M8) grf<1>:f :f :f :f
7	0000,010,010,0,0,010,1,010, 0,0,0,1,0,0000,0,0,010,011	(W) (8 M8) arf<1>:f :f :f :f
8	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,010,011	(W) (8 M8) grf<1>:f :f :f :f
9	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,000,100	(W) (16 M0) grf<1>:f :f :f :f
10	0000,010,010,0,0,010,1,010, 0,0,0,0,0,0000,0,0,000,100	(16 M0) arf<1>:f :f :f :f
11	0000,010,010,1,0,010,1,010, 1,0,0,0,0,0000,0,0,000,100	(16 M0) (sat)grf<1>:f :f :f :f
12	0000,010,010,1,0,010,1,010, 0,0,0,0,0,0000,0,0,100,100	(16 M16) grf<1>:f :f :f :f
13	0000,010,010,0,0,010,1,010, 0,0,0,1,0,0000,0,0,000,100	(W) (16 M0) arf<1>:f :f :f :f
14	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,000,000	(W) (1 M0) grf<1>:f :f :f :f
15	0000,010,010,1,0,010,1,010, 1,0,0,0,0,0000,0,0,000,011	(8 M0) (sat)grf<1>:f :f :f :f
16	0000,010,010,0,0,010,1,010, 0,0,0,1,0,0000,0,0,110,011	(W) (8 M24) arf<1>:f :f :f :f
17	0000,010,010,0,0,010,1,010, 0,0,0,1,0,0000,0,0,100,011	(W) (8 M16) arf<1>:f :f :f :f
18	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,110,011	(W) (8 M24) grf<1>:f :f :f :f
19	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,100,011	(W) (8 M16) grf<1>:f :f :f :f
20	0000,010,010,0,0,010,1,010, 0,0,0,0,0,0000,0,0,100,011	(8 M16) arf<1>:f :f :f :f
21	0000,010,010,0,0,010,1,010, 0,0,0,0,0,0000,0,0,110,011	(8 M24) arf<1>:f :f :f :f
22	0000,010,010,1,0,010,1,010, 0,0,0,0,0,0000,0,0,100,011	(8 M16) grf<1>:f :f :f :f
23	0000,010,010,1,0,010,1,010, 0,0,0,0,0,0000,0,0,110,011	(8 M24) grf<1>:f :f :f :f
24	0000,010,010,0,0,010,1,010, 1,0,0,0,0,0000,0,0,000,100	(16 M0) (sat)arf<1>:f :f :f :f

ControlIndex3	36-Bit Mapping { CondMod[3:0], Src1.SrcType[2:0], Src2.SrcType[2:0], Dst.RegFile[0], Dst.HorzStride[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
25	0000,010,010,1,0,010,1,010, 1,0,0,1,0,0000,0,0,000,100	(W) (16 M0) (sat)grf<1>:f :f :f
26	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,100,100	(W) (16 M16) grf<1>:f :f :f
27	0000,010,010,1,0,010,0,010, 0,0,0,1,0,0000,0,0,000,000	(W) (1 M0) grf<1>:ud :ud :ud
28	0000,010,010,0,0,010,1,010, 0,0,0,0,0,0000,0,0,100,100	(16 M16) arf<1>:f :f :f
29	0000,010,010,1,0,010,1,010, 1,0,0,0,0,0000,0,0,100,100	(16 M16)(sat)grf<1>:f :f :f
30	0000,010,010,1,0,010,1,010, 0,0,0,1,0,0000,0,0,000,010	(W) (4 M0) grf<1>:f :f :f
31	0000,010,010,0,0,010,1,010, 1,0,0,0,0,0000,0,0,000,011	(8 M0) (sat)arf<1>:f :f :f

SourceIndex3 Compact Instruction Field Mappings

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	1,00,1,01,1,00,00, 1,1,00,0,0,00,0,0	grf<0;0> grf<8;1> grf<0>
1	1,00,1,01,1,00,00, 1,0,01,0,0,00,1,0	arf<4;1> grf<8;1> grf<0>
2	1,01,1,01,1,00,00, 1,1,01,0,0,00,1,1	grf<8;1> grf<8;1> grf<1>
3	1,00,1,01,1,00,00, 1,1,01,0,0,00,1,1	grf<8;1> grf<8;1> grf<0>
4	1,01,1,00,0,00,00, 0,1,01,0,0,00,1,1	grf<8;1> grf<0;0> grf<1>
5	1,01,1,01,1,00,00, 1,1,01,0,0,10,1,1	-grf<8;1> grf<8;1> grf<1>
6	1,01,0,01,1,00,00, 1,1,01,0,0,00,1,1	grf<8;1> arf<8;1> grf<1>
7	1,00,0,01,1,00,00, 1,1,00,0,0,00,0,0	grf<0;0> arf<8;1> grf<0>
8	1,01,1,01,1,00,00, 1,1,00,0,0,00,0,0	grf<0;0> grf<8;1> grf<1>

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 }
9	1,01,1,01,1,00,10, 1,1,01,0,0,00,1,1	grf<8;1> grf<8;1> -grf<1>
10	1,01,1,01,1,10,00, 1,1,01,0,0,00,1,1	grf<8;1> - grf<8;1> 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	1,00,0,01,1,00,00, 1,1,01,0,0,00,1,1	grf<8;1> arf<8;1> grf<0>
13	1,00,1,01,1,10,00, 1,1,00,0,0,00,0,0	grf<0;0> - grf<8;1> grf<0>
14	1,00,1,01,1,10,00, 1,1,01,0,0,00,1,1	grf<8;1> - grf<8;1> grf<0>
15	1,00,1,01,1,00,00, 1,1,01,0,0,10,1,1	-grf<8;1> grf<8;1> grf<0>
16	1,00,1,00,0,00,00, 0,1,01,0,0,00,1,1	grf<8;1> grf<0;0> grf<0>
17	1,00,1,01,1,00,00, 1,1,00,0,0,10,0,0	-grf<0;0> grf<8;1> 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>
19	1,01,1,01,1,10,00, 1,1,00,0,0,00,0,0	grf<0;0> - grf<8;1> grf<1>
20	1,00,1,01,1,00,10, 1,1,00,0,0,00,0,0	grf<0;0> grf<8;1> -grf<0>
21	1,01,0,01,1,00,00, 11,00,0,0,00,0,0	grf<0;0> arf<8;1> grf<1>
22	1,00,1,01,1,00,10, 1,1,01,0,0,00,1,1	grf<8;1> grf<8;1> -grf<0>
23	1,01,1,01,1,00,10, 1,1,01,0,0,10,1,1	-grf<8;1> grf<8;1> -grf<1>
24	1,01,0,01,1,00,00, 1,1,01,0,0,10,1,1	-grf<8;1> arf<8;1> grf<1>
25	1,01,1,01,1,10,00, 1,1,01,0,0,10,1,1	-grf<8;1> - grf<8;1> grf<1>
26	1,01,1,00,0,10,00, 0,1,01,0,0,00,1,1	grf<8;1> - grf<0;0> grf<1>
27	1,01,1,00,0,00,10, 0,1,01,0,0,00,1,1	grf<8;1> grf<0;0> -grf<1>

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 }
28	1,01,1,01,1,00,00, 1,1,00,0,0,10,0,0	-grf<0;0> grf<8;1> grf<1>
29	1,01,1,01,1,00,10, 1,1,00,0,0,00,0,0	grf<0;0> grf<8;1> -grf<1>
30	1,00,1,00,0,00,10, 0,1,01,0,0,00,1,1	grf<8;1> grf<0;0> -grf<0>
31	1,01,0,01,1,00,10, 1,1,01,0,0,00,1,1	grf<8;1> arf<8;1> -grf<1>

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
21	00000,00000,10100,00000	.0 .20 .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 }
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=00xxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 110xxxxb 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 1101xxx b.
- 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
0x6C	asr	Component-wise arithmetic shift right	2	1
0x6D	<i>Reserved</i>			
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	<i>Reserved</i>			
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	<i>Reserved</i>			



Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

Flow Control Instructions

Opcode dec hex	Instruction	Description	#src	#dst
32 0x20	jmp	Jump indexed	1	0
33 0x21	brd	Branch - Diverging	1	0
34 0x22	if	If	0/2	0
35 0x23	brc	Branch - Converging	1	-
36 0x24	else	Else	1	0
37 0x25	endif	End if	0	0
38 0x26	<i>Reserved</i>			
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		Instruction	Description	#src	#dst
dec	hex				
49	0x31	send	Send	2	1
53-55	0x35-0x37	<i>Reserved</i>			
56	0x38	math	Math functions for extended math pipeline	1/2	1/2
57-63	0x39-0x3F	<i>Reserved</i>			

Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

Parallel Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
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
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	<i>Reserved</i>			
83	0x53	<i>Reserved</i>			
88	0x58	dp4a	Dot product accumulate	3	1
91	0x5B	mad (fma)	Component-wise floating point mad computation (a multiple-add)	3	1
92	0x5C	<i>Reserved</i>			
93	0x5D	madm (fmam)	Component-wise floating point fused multiply and add for macro operations.	3	1
94	0x5E	<i>Reserved</i>			
95	0x5F	<i>Reserved</i>			

Special Instructions

Instruction
<p>There are two special instructions, <i>illegal</i> and <i>sync</i>.</p> <ul style="list-style-type: none"> <i>Illegal</i> 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 <i>Synchronize</i> 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	<i>Reserved</i>			

Native Instruction BNF

The Backus-Naur Form (BNF) grammar identifies the assembly language syntax, which is native to the hardware. It does not include intelligent defaults, assembler pragmas, etc.

Instruction Groups

<Instruction> ::= <UnaryInstruction>

|<BinaryAcclInstruction>

|<BinaryInstruction>

|<TrilInstruction>

|<JumpInstruction>

|<BranchLoopInstruction>

|<ElseInstruction>

|<BreakInstruction>

|<MaskControllInstruction

|<TrilInstruction2>

|<CallInstruction>

|<BranchConvIntruction>

|<BranchDivInstruction>

|<MathInstruction>

|<SynclInstruction>

|<SpecialInstruction>

<UnaryInstruction> ::= <Predicate> <UnaryInst> <ExecSize> dst <SrcAcclmm> <InstOptions>

<UnaryInst> ::= <UnaryOp> <ConditionalModifier> <Saturate>

<UnaryOp> ::= "mov" | "frc" | "rndu" | "rddd" | "rnde" | "rndz" | "not" | "lzd"

<BinaryInstruction> ::= <Predicate> <BinaryInst> <ExecSize> dst <Src> <SrcImm> <InstOptions>

<BinaryInst> ::= <BinaryOp> <ConditionalModifier> <Saturate>

<BinaryOp> ::= "mul" | "mac" | "mach" | "line" | "pln"

"sad2" | "sada2" | "dp4" | "dph" | "dp3" | "dp2" | "lrp" | "bfi" | "addc" | "subb"

<BinaryAcclInstruction> ::= <Predicate> <BinaryAcclInst> <ExecSize> dst <SrcAcc> <SrcImm>
<InstrOptions>

<BinaryAcclInst> ::= <BinaryAccOp> <ConditionalModifier> <Saturate>

<BinaryAccOp> ::= "avg" | "add" | "sel"

"and" | "or" | "xor"

"shr" | "shl" | "asr"

"cmp" | "cmpn"



<TriInstruction> ::= <Predicate> <TriInst> <ExecSize> <PostDst> <CurrDst> <TriSrc> <MsgDesc>
<InstOptions>
<TriInst> ::= <TriOp> <ConditionalModifier> <Saturate>
<TriOp> ::= **"send"**
<TriInstruction2> ::= <Predicate> <TriInst2> <ExecSize> dst <Src> <Src> <Src> <InstOptions>
<TriInst2> ::= <TriOp> <ConditionalModifier> <Saturate>
, <TriOp> ::= **"bfe" | "bfi2" | "mad"**
<BranchConvInstruction> ::= <Predicate> <BranchConvOp> <ExecSize> <RelativeLocation2>
<BranchConvOp> ::= **"brc"**
<BranchDivInstruction> ::= <Predicate> <BranchDivOp> <ExecSize> <RelativeLocation3>
<BranchDivOp> ::= **"brd"**
<CallInstruction> ::= <Predicate> <CallOp> <ExecSize> dst <RelativeLocation2>
<CallOp> ::= **"call" | "CALLA"**
<MathInstruction> ::= <Predicate> <MathInst> <ExecSize> <Dst> <Src> <Src> <FC>
<MathInst> ::= <MathOp> <Saturate>
<MathOp> ::= **"math"**
<FC> ::= **"INV" | "LOG" | "EXP" | "SQRT" | "RSQ" | "POW" | "SIN" | "COS" | "INT DIV"**
<JumpInstruction> ::= <JumpOp> <RelativeLocation2>
<JumpOp> ::= **"jmp" | "jmpi"**
<BranchLoopInstruction> ::= <Predicate> <BranchLoopOp> <RelativeLocation>
<BranchLoopOp> ::= **"if" | "iff" | "while"**
<ElseInstruction> ::= <ElseOp> <RelativeLocation>
<ElseOp> ::= **"else"**
<BreakInstruction> ::= <Predicate> <BreakOp> <LocationStackCtrl>
<BreakOp> ::= **"break" | "cont" | "halt"**
<SyncInstruction> ::= <Predicate> <SyncOp> <NotifyReg>
<SyncOp> ::= **"wait"**
<SpecialInstruction> ::= **"do" | "endif" | "nop" | "illegal"**

Destination Register

dst ::= <DstOperand>

|<DstOperandEx>

<DstOperand> ::= <DstReg> <DstRegion> <WriteMask> <DstType>

<DstOperandEx> ::= <AccReg> <DstRegion> <DstType>

|<FlagReg> <DstRegion> <DstType>

|<AddrReg> <DstRegion> <DstType>

|<MaskReg> <DstRegion> <DstType>

|<MaskStackReg>

|<ControlReg>

|<IPReg>

|<NullReg>

| <ChannelEnableReg>

|<ThreadControlReg>

|<PerformanceReg>

<DstReg> ::= <DirectGenReg> | <IndirectGenReg>

|<DirectMsgReg> | <IndirectMsgReg>

<PostDst> ::= <PostDstReg> <DstRegion> <WriteMask> <DstType>

|<NullReg>

<PostDstReg> ::= <DirectGenReg> | <IndirectGenReg>

<CurrDst> ::= <DirectAlignedMsgReg>



Source Register

Source with Accumulator Access and with Immediate

<SrcAcclmm> ::= <SrcAcc>
| <Imm32> <SrcImmType>
<SrcAcc> ::= <DirectSrcAccOperand>
| <IndirectSrcOperand>
<DirectSrcAccOperand> ::= <DirectSrcOperand>
| <SrcArcOperandEx>
| <AccReg> <SrcType>
<SrcArcOperandEx> ::= <FlagReg> <Region> <SrcType>
| <AddrReg> <Region> <SrcType>
| <ControlReg>
| <StateReg>
| <NotifyReg>
| <IPReg>
| <NullReg>
| <ChannelEnableReg>
| <ThreadControlReg>
| <PerformanceReg>
<IndirectSrcOperand> ::= <SrcModifier> <IndirectGenReg> <IndirectRegion> <Swizzle> <SrcType>

Source without Accumulator Access

<Src> ::= <DirectSrcOperand>
| <IndirectSrcOperand>
<DirectSrcOperand> ::= <SrcModifier> <DirectGenReg> <Region> <Swizzle> <SrcType>
| <SrcArcOperandEx>
<TriSrc> ::= <SrcModifier> <DirectGenReg> <Region> <Swizzle> <SrcType>
| <NullReg>
<MsgDesc> ::= <ImmDesc>
| <Reg32>
<Reg32> ::= <DirectGenReg> <Region> <SrcType>

Source without Accumulator Access or IP Access

<SrcImm> ::= <DirectSrcOperand>

| <Imm32> <SrcImmType>

Address Registers

<AddrParam> ::= <AddrReg> <ImmAddrOffset>

<ImmAddrOffset> ::= ""

| ", " <ImmAddrNum>

Register Files and Register Numbers

Note: The recommended instruction syntax uses subregister numbers within the GRF in units of actual data element size, corresponding to the data type used. For example for the F (Float) type, the assembler syntax uses subregister numbers 0 to 7, corresponding to subregister byte addresses of 0 to 28 in steps of 4, the element size.

<DirectGenReg> ::= <GenRegFile> <GenRegNum> <GenSubRegNum>

<IndirectGenReg> ::= <GenRegFile> "[" <AddrParam> "]"

<GenRegFile> ::= "r"

<GenRegNum> ::= "0" ... "127"

<GenSubRegNum> ::= = ""

| ".0" ... ".3" //incase of DF

| ".0" ... ".7"

| ".0" ... ".15"

| ".0" ... ".31"

<DirectMsgReg> ::= <DirectAlignedMsgReg> <MsgSubRegNum>

<DirectAlignedMsgReg> ::= <MsgRegFile> <MsgRegNum>

<IndirectMsgReg> ::= <MsgRegFile> "[" <AddrParam> "]"

<MsgRegFile> ::= "m"

<MsgRegNum> ::= "0" ... "15"

<MsgSubRegNum> ::= <GenSubRegNum>

<AddrReg> ::= <AddrRegFile> <AddrSubRegNum>

<AddrRegFile> ::= "a0"

<AddrSubRegNum> ::= = ""

| ".0" ... ".7"

<AccReg> ::= "acc" <AccRegNum> <AccSubRegNum>



<AccRegNum> ::= "0" | "1"
<AccSubRegNum> ::= <GenSubRegNum>
<FlagReg> ::= "f" <FlagRegNum> <FlagSubRegNum>
<FlagRegNum> ::= "0" | "1"
<FlagReg> ::= "f0" <FlagSubRegNum>
<FlagSubRegNum> ::= ""
| ".0"..."1"
<NotifyReg> ::= "n" <NotifyRegNum>
<NotifyRegNum> ::= "0"..."2"
<StateReg> ::= "sr0" <StateSubRegNum>
<StateSubRegNum> ::= ".0"..."1"
<ControlReg> ::= "cr0" <ControlSubRegNum>
<ControlSubRegNum> ::= ".0"..."2"
<IPReg> ::= "ip"
<NullReg> ::= "null"
<ThreadControlReg> ::= "tdr0" <ThreadCntrlSubRegNum>
<ThreadCntrlSubRegNum> ::= ".0"..."7"
<PerformanceReg> ::= "tm0"
<ChannelEnableReg> ::= "ce0.0"

Relative Location and Stack Control

<RelativeLocation> ::= <imm16>
<RelativeLocation2> ::= <imm32> | <reg32>
<RelativeLocation3> ::= <imm16> | <reg32>
<LocationStackCtrl> ::= <imm32>

Regions

<DstRegion> ::= "<" <HorzStride> ">"
<IndirectRegion> ::= <Region> | <RegionWH> | <RegionV>
<Region> ::= "<" <VertStride> ";" <Width> "," <HorzStride> ">"
<RegionWH> ::= "<" <Width> "," <HorzStride> ">"
<RegionV> ::= "<" <VertStride> ">"
<VertStride> ::= "0" | "1" | "2" | "4" | "8" | "16" | "32"

<Width> ::= "1" | "2" | "4" | "8" | "16"

<HorzStride> ::= "0" | "1" | "2" | "4"

Types

SrcType
<SrcType> ::= ":df" ":f" ":ud" ":d" ":uw" ":w" ":ub" ":b" ":uq" ":q" ":hf"

<SrcImmType> ::= <SrcType> | ":v" | ":vf" | ":uv"

<DstType> ::= <SrcType>

Write Mask

<WriteMask> ::= ""

| "." "x" | "." "y" | "." "z" | "." "w"

| "." "xy" | "." "xz" | "." "xw" | "." "yz" | "." "yw" | "." "zw"

| "." "xyz" | "." "xyw" | "." "xzw" | "." "yzw"

| "." "xyzw"

Swizzle Control

<Swizzle> ::= ""

| "." <ChanSel>

| "." <ChanSel> <ChanSel> <ChanSel> <ChanSel>

<ChanSel> ::= "x" | "y" | "z" | "w"

Immediate Values

<ImmAddrNum> ::= "-512" ... "511"

<Imm64> ::= "0.0" ... "+/-1.0*2-1024...1023" | "0" ... "264-1" | "-263" ... "263-1"

<Imm32> ::= "0.0" ... "+/-1.0*2-128...127" | "0" ... "2³²-1" | "-2³¹" ... "2³¹-1"

<Imm16> ::= "0" ... "2¹⁶-1" | "-2¹⁵" ... "2¹⁵-1"

<ImmDesc> ::= "0" ... "2³²-1"



Predication and Modifiers

Instruction Predication

<Predicate> ::= ""

| "(" <PredState> <FlagReg> <PredCntrl> ")"

<PredState> ::= ""

| "+"

| "-"

<PredCntrl> ::= ""

| ".x" | ".y" | ".z" | ".w"

| ".any2h" | ".all2h"

| ".any4h" | ".all4h"

| ".any8h" | ".all8h"

| ".any16h" | ".all16h"

| ".anyv" | ".allv"

| ".any32h" | ".all32h"

Source Modification

<SrcModifier> ::= ""

| "-"

| "(abs)"

| "-" "(abs)"

Instruction Modification

<ConditionalModifier> ::= ""

| <CondMod> "." <FlagReg>

<CondMod> ::= ".z" | ".e" | ".nz" | ".ne" | ".g" | ".ge" | ".l" | ".le" | ".o" | ".r" | ".u"

<Saturate> ::= ""

| ".sat"

Execution Size

<ExecSize> ::= "(" <NumChannels> ")"

<NumChannels> ::= "1" | "2" | "4" | "8" | "16" | "32"

Instruction Options

```

<InstOptions> ::= ""
| "{" <InstOption> "}"
| "{" <InstOption> <InstOptionEx> "}"
<InstOptionEx> ::= ""
| "," <InstOption> <InstOptionEx>
<InstOption> ::= <AccessMode>
| <AccWrCtrl>
| <ComprCtrl>
| <DependencyCtrl>
| <MaskCtrl>
| <SendCtrl>
| <ThreadCtrl>
<AccessMode> ::= "Align1" | "Align16"
<AccWrCtrl> ::= "AccWrEn"
<ComprCtrl> ::= "SecHalf" | "Compr"
<DependencyCtrl> ::= "NoDDChk" | "NoDDClr"
<MaskCtrl> ::= "NoMask"
<SendCtrl> ::= "EOT"
<ThreadCtrl> ::= "Switch"
| "Atomic"

```

Note for Assembler: Compression control "**Compr**" has a direct map to the binary instruction word. It may be omitted if the Assembler can determine whether an instruction is compressible.



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.

SrcMod Information
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?
<i>add</i>	40	Addition	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>addc</i>	4E	Integer Addition with Carry	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
<i>and</i>	65	Logic And	Move and Logic	2	Y	**	Int	Int	N	Equality only
<i>asr</i>	6C	Arithmetic Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
<i>avg</i>	42	Average	Parallel Arithmetic	2	Y	Y	B, UB W, UW D, UD	B, UB W, UW D, UD	Y	Y
<i>bfe</i>	78	Bit Field Extract	Move and Logic	3	Y	N	UD, D	UD, D	N	N
<i>bfi1</i>	79	Bit Field Insert 1	Move and Logic	2	Y	N	UD, D	UD, D	N	N
<i>bfi2</i>	7A	Bit Field Insert 2	Move and Logic	3	Y	N	UD, D	UD, D	N	N

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
<i>bprev</i>	77	Bit Field Reverse	Move and Logic	1	Y	N	UD	UD	N	N
<i>brc</i>	23	Branch Converging	Flow Control	0 or 1	Y	N	D		N	N
<i>brd</i>	21	Branch Diverging	Flow Control	0 or 1	Y	N	D		N	N
<i>break</i>	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?
<i>call</i>	2C	Call	Flow Control	0	Y	N		D, UD	N	N
<i>calla</i>	2B	Call Absolute	Flow Control	0	Y	N		D, UD	N	N
<i>cbit</i>	4D	Count Bits Set	Move and Logic	1	Y	N	UB, UW, UD	UD	N	N
<i>cmp</i>	70	Compare	Move and Logic	2	Y	Y	*	*	N	Y
<i>cmpn</i>	71	Compare NaN	Move and Logic	2	Y	Y	*	*	N	Y
<i>cont</i>	29	Continue	Flow Control	0	Y	N			N	N
<i>csel</i>	72	Conditional Select	Move and Logic	3	N	Y	F	F	Y	N
<i>dp4a</i>	58	Dot Product 4 Accumulate	Vector Arithmetic	2	Y	Y	F	F	Y	Y
<i>else</i>	24	Else	Flow Control	0	N	N			N	N
<i>endif</i>	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?
<i>fbh</i>	4B	Find First Bit from MSB Side	Move and Logic	1	Y	N	D, UD	UD	N	N
<i>fbl</i>	4C	Find First Bit from LSB Side	Move and Logic	1	Y	N	UD	UD	N	N
<i>frc</i>	43	Fraction	Parallel Arithmetic	1	Y	Y	F	F	N	Y
<i>goto</i>	2E	Goto	Flow Control	0	Y	N			N	N

Mnem.	Hex Opcode	Name	Group	Srcs	Pred?	SrcMod	Src Types	Dst Type	Sat?	CondMod?
<i>halt</i>	2A	Halt	Flow Control	0	Y	N			N	N
<i>if</i>	22	If	Flow Control	0	Y	N			N	N
<i>illegal</i>	00	Illegal	Special	0	N	N			N	N
<i>jmp</i>	20	Jump Indexed	Flow Control	1	Y	N	D		N	N
<i>join</i>	2F	Join	Flow Control	0	Y	N			N	N
<i>line</i>	59	Line	Vector Arithmetic	2	Y	Y	F	F	Y	Y
<i>lzd</i>	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?
<i>mac</i>	48	Multiply Accumulate	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>mach</i>	49	Multiply Accumulate High	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>mad</i>	5B	Multiply Add	Vector Arithmetic	3	Y	Y	*	*	Y	N
<i>madm</i>	5D	Multiply Add for Macro	Vector Arithmetic	3	Y	Y	*	*	Y	N
<i>math</i>	38	Extended Math Function	Miscellaneous	2	Y	N	*	*	Y	N
<i>mov</i>	01	Move	Move and Logic	1	Y	Y	*	*	Y	Y
<i>mov</i>	61	Move	Move and Logic	1	Y	Y	*	*	Y	Y
<i>movi</i>	63	Move Indexed	Move and Logic	1	Y	Y	*	*	Y	N
<i>mul</i>	41	Multiply	Parallel Arithmetic	2	Y	Y	*	*	Y	Y
<i>nop</i>	60	No Operation	Move and Logic	0	N	N			N	N
<i>not</i>	64	Logic Not	Move and Logic	1	Y	**	Int	Int	N	Equality only
<i>or</i>	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?
<i>ret</i>	2D	Return	Flow Control	1	Y	N	D, UD		N	N
<i>rndd</i>	45	Round Down	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>rnde</i>	46	Round to Nearest or Even	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>rndu</i>	44	Round Up	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>rndz</i>	47	Round to Zero	Parallel Arithmetic	1	Y	Y	F	F	Y	Y
<i>sad2</i>	50	Sum of Absolute Difference 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
<i>sada2</i>	51	Sum of Absolute Difference Accumulate 2	Vector Arithmetic	2	Y	Y	B, UB	W, UW	Y	Y
<i>sel</i>	62	Select	Move and Logic	2	Y	Y	*	*	Y	Y
<i>send</i>	31	Send Message	Miscellaneous	2	Y	N	*	*	N	N
<i>sendc</i>	32	Send Message Conditional	Miscellaneous	2	Y	N	*	*	N	N
<i>shl</i>	69	Shift Left	Move and Logic	2	Y	Y	Int	Int	Y	Y
<i>shr</i>	68	Shift Right	Move and Logic	2	Y	Y	Int	Int	Y	Y
<i>smov</i>	6A	Scattered Move	Move and Logic	1	Y	N	*	*	N	N
<i>subb</i>	4F	Integer Subtraction with Borrow	Parallel Arithmetic	2	Y	N	UD	UD	N	Y
<i>sync</i>	01	Synchronize	Special	1	Y	N	*	*	N	N
<i>while</i>	27	While	Flow Control	0	Y	N			N	N
<i>xor</i>	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.

Accumulator Restrictions

Instructions	Accumulator Restrictions
<i>add</i>	No restrictions.
<i>addc</i>	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.
<i>and</i>	Source modifier is not allowed if source is an accumulator.
<i>asr</i> <i>avg</i>	No restrictions.
<i>bfe</i> <i>bfi1</i> <i>bfi2</i> <i>bfrex</i> <i>cbit</i>	No accumulator access, implicit or explicit.
<i>cmp</i>	No restrictions.
<i>cmpn</i>	No restrictions.
<i>csel</i>	No restrictions.
<i>fbh</i> <i>fbl</i>	No accumulator access, implicit or explicit.
<i>frc</i>	No restrictions.
<i>line</i>	Source operands cannot be accumulators.
<i>lzd</i>	Accumulator cannot be source, implicit or explicit.
<i>mac</i>	Accumulator is an implicit source and thus cannot be an explicit source operand.
<i>mac</i>	Integer source operands cannot be explicit accumulators.
<i>mach</i>	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.
<i>mad</i>	Integer source operands cannot be accumulators.
<i>madm</i>	No explicit accumulator access because this is a three-source instruction. AccWrEn is allowed for implicitly updating the accumulator.
<i>math</i>	No accumulator access, implicit or explicit.
<i>mov</i>	An accumulator can be a source or destination operand but not both.
<i>movi</i>	Source operands cannot be accumulators.



Instructions	Accumulator Restrictions
<i>mul</i>	Integer source operands cannot be accumulators.
<i>not</i> <i>or</i>	Source modifier is not allowed if source is an accumulator.
<i>rndd</i> <i>rnde</i> <i>rndu</i> <i>rndz</i>	No restrictions.
<i>sel</i>	No restrictions.
<i>shl</i>	Accumulator cannot be destination, implicit or explicit.
<i>shr</i>	No restrictions.
<i>smov</i>	No restrictions.
<i>subb</i>	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.
<i>xor</i>	Source modifier is not allowed if source is an accumulator.

Native Instruction BNF

Describes Syntax supported by IGA.

Grammar

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	->	<i>PredWrEn? Mnemonic ExecInfo FlagModifier? Operands InstOptions?</i>	
where			
e.g. mov or math.sqrt			
<i>Mnemonic</i>	->	<i>Ident math . MathFC</i>	
e.g. sqrt in math.sqrt ... math function controls extend the opcode; e.g. math.inv ...			
<i>MathFC</i>	->	inv	
		log	
		exp	
		sqrt	
		rsqrt	
		pow	
		sin	
		cos	
		idiv	
		iqot	
		irem	
		fdiv	
		invm	
		rsqtm	
Execution mask part e.g. (16 M16)			
<i>ExecInfo</i>	->	<i>(ExecWidth EMaskOff)</i>	
where			
<i>ExecWidth</i>	->	1 2 4 8 16 32	
the emask			
<i>EMaskOff</i>	->	M0 M4 ... M28	
Instruction write-enable (formly NoMask) as well as predication Examples: (W), (f1.0), or (W~f1.1)			
<i>PredWrEn</i>	->	<i>(WrEn) (Pred) (WrEn & Pred)</i>	
where			
<i>WrEn</i>	->	W	
<i>Pred</i>	->	<i>PredNegate? FlagRegRef PredMask?</i>	

<i>PredNegate</i>	->	~
see PredCtrl enum 3D-Media-GPGPU Engine EU Overview ISA Introduction Instruction Set Reference EUISA Enumerations PredCtrl		
<i>PredMask</i>	->	. <i>PredFunc</i>
where		
<i>PredFunc</i>	->	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	->	(<i>Func</i>) <i>FlagRegRef</i>	
where			
<i>Func</i>	->	lt	
		le	
		gt	
		ge	
		eq	
		ne	
		ov	overflow
		un	unordered (NaN)
		gz	less than 0; only supported on sel instructions
		lz	greater than 0; only supported on sel instructions

Operands	->	<i>OperandsEmpty</i>	e.g. nop, illegal, ...
		<i>OperandsUnary</i>	e.g. mov, not, ... [ERROR: unclosed literal escape with \@]
		<i>OperandsBinary</i>	e.g. and, add, mul, ... [ERROR: unclosed literal escape with \@]
		<i>OperandsTernary</i>	e.g. mad, bfi2

		<i>OperandsSingleSource</i>	e.g. wait, jmpi, ...
		<i>OperandsDoubleSource</i>	e.g. break, cont, if, else (branch control is a sub function)
		<i>OperandsSend</i>	send or sendc
		<i>OperandsSplitSend</i>	sends or sendsc

OperandsEmpty	->	

OperandsUnary	->	<i>DstOperand SrcOperand</i>

OperandsBinary	->	<i>DstOperand SrcOperand SrcOperand</i>	

OperandsTernary	->	<i>DstOperand SrcOperand SrcOperand SrcOperand</i>		

OperandsSingleSource	->	<i>SrcOperand</i>

OperandsDoubleSource	->	<i>SrcOperand SrcOperand</i>

OperandsSend	->	<i>DstOperand SrcOperand ExDescOp DescOp</i>	

OperandsSplitSend	->	<i>DstOperand SrcOperand SrcOperand ExDescOp DescOp</i>		

ExDescOp	->	<i>AddrRegRef IntLit</i>

DescOp	->	<i>AddrRegRef IntLit</i>

Destination operands			
DstOperand	-	<i>DstOpDirect</i>	direct register access; e.g. r14.3<1> or sr0.1
	>		
		<i>DstOpIndirect</i>	indirect access; e.g. r[a0.4,16]<1>
where			

<i>DstOpDirect</i>	- >	<i>DstModif?</i> Register <i>DstRegion?</i>	
<i>DstOpIndirect</i>	- >	<i>DstModif?</i> r [<i>AddrRegName</i> .] <i>DstRegion?</i>	
<i>DstModif</i>	- >	(sat)	saturation modifier
<i>DstRegion</i>	- >	< <i>DstHzStride</i> >	
where			
<i>DstHzStride</i>	- >	1 2 4	

Source operands -- Semantic checks for regioning disallow regions on certain registers (e.g. cannot region ce)			
SrcOperand	- >	<i>SrcOpDirect</i>	direct register access; e.g. r13.0<8;8,1>
		<i>SrcOpIndirect</i>	indirect register access; e.g. r[a0.0,16]<1,0>
		<i>SrcOpImm</i>	source immediate operand; e.g. 13:ud, 0x155:q, 1.2:f
where			
<i>SrcOpDirect</i>	- >	<i>SrcModifier?</i> Register <i>SrcRegion?</i> <i>SrcRegType?</i>	
<i>SrcOpIndirect</i>	- >	<i>SrcModifier?</i> r [<i>IndexReg</i> (, <i>ConstExpr</i>)] <i>SrcRegion?</i> <i>SrcRegType?</i>	
<i>SrcOpImm</i>	- >	<i>ConstExpr</i> <i>SrcImmType</i>	
We semantically enforced that ~ applies only to bitwise ops (e.g. and, or, ...)			
<i>SrcModifier</i>	- >	~	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			
<i>SrcRegion</i>	- >	<i>SrcRegionVWH</i> <i>SrcRegionWH</i> <i>SrcRegionVH</i> <i>SrcRegionH</i>	
<i>SrcRegionVWH</i>	- >	< <i>VtStride</i> ; <i>Width</i> , <i>HzStride</i> >	
<i>SrcRegionWH</i>	- >	< <i>Width</i> , <i>HzStride</i> >	for indirect only (uses VxH/Vx1 mode)
<i>SrcRegionVH</i>	- >	< <i>VtStride</i> ; <i>HzStride</i> >	ternary src0 or src1 region
<i>SrcRegionH</i>	- >	< <i>HzStride</i> >	ternary src2 region

	>		
<i>VtStride</i>	-	1	
	>		
		2	
		4	
		8	
		16	
		32	the number of elements to advance the row after stepping Width times
<i>Width</i>	-	1	
	>		
		2	
		4	
		8	
		16	the number elements per row (before advancing vert stride)
<i>HzStride</i>	-	0	
	>		
		1	
		2	
		4	the number of elements to step horizontally

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	->	<i>RegName (. SubReg)?</i>	
where			
GRF and ARF registers. See: BSpec: 3D-Media-GPGPU Engine EU Overview Registers and Register Regions			
<i>RegName</i>	->	<i>Names</i>	
where			
<i>Names</i>	->	<i>AddrRegName</i>	address register
		<i>acc0</i>	
		...	
		<i>acc11</i>	up to <i>acc1</i> previously and <i>acc9</i>
		<i>FlagRegName</i>	f0 or f1
		<i>fc0</i>	
		...	
		<i>fc4</i>	flow control registers
		<i>ce</i>	channel enable register

		cr0	
		ip	instruction pointer
		n0	
		n1	
		msg0	
		...	
		msg31	
		null	the null register
		r0	
		...	
		r127	
		sp	stack pointer
		spl	stack-pointer limit (formerly sp_limit)
		sr0	
		sr1	
		tdr	
		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 ...			
<i>SubReg</i>	->	<i>IntLit</i>	

A reference to a flag register. Examples: f0.0 or f1.1			
FlagRegRef	->	<i>FlagRegName (. FlagSubReg)?</i>	
where			
<i>FlagSubReg</i>	->	0 1	

FlagRegName	->	f0 f1	

An address register reference; e.g. a0.3 (sometimes called the "index register")			
AddrRegRef	->	<i>AddrRegName (. AddrSubReg)?</i>	
where			
<i>AddrSubReg</i>	->	0 ... 15	

AddrRegName	->	a0	

Destination types are a subset of the source types as only the Packed vectors types are not			
<i>DstType</i>	->	:b	
		:ub	
		:w	
		:uw	
		:d	
		:ud	
		:q	
		:uq	
		:hf	
		:f	
		:df	

Source types are the same as the destination types with the inclusion of the packed vector types.			
<i>SrcType</i>	->	<i>DstType</i>	
		:uv	packed unsigned int
		:v	packed signed int
		:vf	packed float

Expressions. Constant expressions are evaluated by the assembler with host semantics.			
<i>ConstExpr</i>	->	<i>IntLit</i>	integer literal
		<i>FltLit</i>	floating point literal
		<i>Ident</i>	reference to a symbolic constant
		<i>UnOp ConstExpr</i>	unary constant expressions (e.g. $\sim((1NUM_BITS)-1)$)
		<i>ConstExpr BinOp ConstExpr</i>	binary constant expressions (1+2 or 1BIT_INDEX)
		(<i>ConstExpr</i>)	constant grouping expression (1 + 2)
		<i>LabelRef</i>	a reference to a label definition

Unary operators			
<i>UnOp</i>	->	~	one's complement
		-	two's complement

Binary operators			
<i>BinOp</i>	->	*	

		/	
		%	multiplicative operators
		+	
		-	additive operators
		<<	
		>>	
		>>>	bit-shift operators
		&	bitwise AND
			bitwise OR
		^	bitwise XOR

Identifiers		
Ident	->	[A-Za-z_][A-Za-z0-9_]*

An integer literal. For each, _ is an optional lexical separator for readability			
IntLit	->	<i>IntLitDec</i>	decimal literal e.g. 13, 14, 4000, or 4_000
		<i>IntLitHex</i>	hex literal e.g. 0xFE or 0xABCD_EFGH
		<i>IntLitBin</i>	binary literal e.g. 0b1011 would be 11
where			
<i>IntLitDec</i>	->	[0-9][_0-9]*	
<i>IntLitHex</i>	->	0x[0-9a-fA-F]?[_0-9a-fA-F]*	
<i>IntLitBin</i>	->	0b[01][_01]*	

A floating point literal			
FltLit	- >	<i>FltLitDec</i>	decimal 3.1414
		<i>FltLitSci</i>	scientific 6.02e23 or 6.67384e-11
		<i>inf</i>	special symbol for infinity (0x...)
		<i>nan</i>	special symbol for not a number
where			
<i>FltLitDec</i>	- >	$([0-9][_0-9]*)?.[0-9][_0-9]^* ([0-9][_0-9]^*)?.([0-9][_0-9]^*)?$	
<i>FltLitSci</i>	- >	$([0-9][_0-9]*)?.[0-9][_0-9]^*e([0-9][_0-9]^*)$	

InstOptions	->	{ (InstOpt (, InstOpt)*)? }	
where			
<i>InstOpt</i>	->	AccWrEn	
		Atomic	
		Breakpoint	
		Compacted	
		EOT	
		NoDDChk	
		NoDDClr	
		NoPreempt	
		NoSrcDep	
		Switch	

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).

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	-	<i>PredWrEn? Mnemonic ExeInfo</i>
	>	<i>FlagModifier? Operands InstOptions?</i>
where		
<i>Mnemonic</i>	-	<i>Ident</i>
	>	E.g. <i>mov</i> or <i>add</i>
		<i>math . MathFC</i>
		E.g. <i>math.inv</i>
		<i>send . SFID</i>
		E.g. <i>send.dc0</i>
		<i>bfm . BfmFC</i>
		E.g. <i>bfm.0xE7</i>
		<i>dpas . DpasFC</i>
		E.g. <i>dpas.4x4</i>
e.g. <i>sqrt</i> in <i>math.sqrt</i> ... <i>math</i> function controls extend the opcode; e.g. <i>math.inv</i> ...		
<i>MathFC</i>	-	<i>inv</i>
	>	
		<i>log</i>
		<i>exp</i>
		<i>sqrt</i>
		<i>rsqrt</i>
		<i>pow</i>
		<i>sin</i>
		<i>cos</i>
		<i>idiv</i>
		<i>iqot</i>
		<i>irem</i>
		<i>fdiv</i>
		<i>invm</i>
		<i>rsqtm</i>
e.g. <i>dco</i> in <i>send.dco</i> ... <i>send</i> function id determines what <i>send</i> type it is		
<i>SFID</i>	-	<i>dc0</i>
	>	
	-	<i>dc1</i>
	>	
	-	<i>dc2</i>
	>	
	-	<i>dcr0</i>
	>	
	-	<i>gtwy</i>
	>	
	-	<i>null</i>
	>	
	-	<i>pixi</i>
	>	

	>		
	-	rc	
	>		
	-	smp1	
	>		
	-	ts	
	>		
	-	urb	
	>		
	-	vme	
	>		
	-	cre	
	>		
<i>BfnFC</i>	-	See BooleanFuncCtrl enumeration for a list of legal symbols	e.g. bfn.0xE2
	>		
<i>DpasFC</i>	-	See the SystolicFC enumeration for a list of legal subop symbols	e.g. dpas.4x8 is a DPAS with systolic depth 4 and repeat count 8
	>		
Execution mask part e.g. (16 M16)			
<i>ExecInfo</i>	-	(ExecWidth EMaskOff)	
	>		
where			
<i>ExecWidth</i>	-	1 2 4 8 16 32	
	>		
the emask			
<i>EMaskOff</i>	-	M0 M4 ... M28	
	>		
Instruction write-enable (formly NoMask) as well as predication Examples: (W), (f1.0), or (W~f1.1)			
<i>PredWrEn</i>	-	(WrEn) (Pred) (WrEn & Pred)	
	>		
where			
<i>WrEn</i>	-	W	
	>		
<i>Pred</i>	-	PredNegate? FlagRegRef PredMask?	
	>		
<i>PredNegate</i>	-	~	
	>		
see PredCtrl enum 3D-Media-GPGPU Engine EU Overview ISA Introduction Instruction Set Reference EUISA Enumerations PredCtrl			
<i>PredMask</i>	-	. PredFunc	
	>		
where			
<i>PredFunc</i>	-	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 than or equal
		gt	greater than
		ge	greater than or equal
		eq	equal
		ne	not equal
		ov	overflow
		un	unordered (NaN)
		gz	less than 0; only supported on sel instructions
		lz	greather 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, bfn, dpas, ...
		OperandsSingleSource	e.g. wait, jmp, ...
		OperandsDoubleSource	e.g. break, cont, if, else (branch control is a sub function)
		OperandsSend	sends or sendsc

OperandsEmpty	->	
----------------------	----	--

OperandsUnary	->	<i>DstOperand SrcOperand</i>
----------------------	----	------------------------------

OperandsBinary	->	<i>DstOperand SrcOperand SrcOperand</i>
-----------------------	----	---

OperandsTernary	->	<i>DstOperand SrcOperand SrcOperand SrcOperand</i>
------------------------	----	--

OperandsSingleSource	->	<i>SrcOperand</i>
-----------------------------	----	-------------------

OperandsDoubleSource	->	<i>SrcOperand SrcOperand</i>
-----------------------------	----	------------------------------

OperandsSend	->	<i>DstOperand SrcOperand SrcOperand ExDescOp DescOp</i>
---------------------	----	---

ExDescOp	->	<i>AddrRegRef IntLit</i>
-----------------	----	----------------------------

DescOp	->	<i>AddrRegRef IntLit</i>
---------------	----	----------------------------

Destination operands			
DstOperand	-	<i>DstOpDirect</i>	direct register access; e.g. r14.3<1> or sr0.1
	>		
		<i>DstOpIndirect</i>	indirect access; e.g. r[a0.4,16]<1>
where			
<i>DstOpDirect</i>	-	<i>DstModif? Register DstRegion? DstType?</i>	
	>		
<i>DstOpIndirect</i>	-	<i>DstModif? r [AddrRegName .]</i>	
	>	<i>DstRegion?</i>	
<i>DstModif</i>	-	(sat)	saturation modifier
	>		
<i>DstRegion</i>	-	< <i>DstHzStride</i> >	
	>		
where			

<i>DstHzStride</i>	-	1 2 4
	>	

Source operands -- Semantic checks for regioning disallow regions on certain registers (e.g. cannot region ce)			
SrcOperand	-	<i>SrcOpDirect</i>	direct register access; e.g. r13.0<8;8,1>
	>		
		<i>SrcOpIndirect</i>	indirect register access; e.g. r[a0.0,16]<1,0>
		<i>SrcOpImm</i>	source immediate operand; e.g. 13:ud, 0x155:q, 1.2:f
where			
<i>SrcOpDirect</i>	-	<i>SrcModifier?</i> Register <i>SrcRegion?</i>	
	>	<i>SrcRegType?</i>	
<i>SrcOpIndirect</i>	-	<i>SrcModifier?</i> r [<i>IndexReg</i> (, <i>ConstExpr</i>)]	
	>	<i>SrcRegion?</i> <i>SrcRegType?</i>	
<i>SrcOpImm</i>	-	<i>ConstExpr SrcImmType</i>	
	>		
We semantically enforced that ~ applies only to bitwise ops (e.g. and, or, ...)			
<i>SrcModifier</i>	-	~	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			
<i>SrcRegion</i>	-	<i>SrcRegionVWH</i> <i>SrcRegionWH</i>	
	>	<i>SrcRegionVH</i> <i>SrcRegionH</i>	
<i>SrcRegionVWH</i>	-	< <i>VtStride</i> ; <i>Width</i> , <i>HzStride</i> >	
	>		
<i>SrcRegionWH</i>	-	< <i>Width</i> , <i>HzStride</i> >	for indirect only (uses VxH/Vx1 mode)
	>		
<i>SrcRegionVH</i>	-	< <i>VtStride</i> ; <i>HzStride</i> >	ternary src0 or src1 region
	>		
<i>SrcRegionH</i>	-	< <i>HzStride</i> >	ternary src2 region
	>		
<i>VtStride</i>	-	1	
	>		
		2	
		4	
		8	
		16	
		32	the number of elements to advance the row after stepping Width times

<i>Width</i>	-	1	
	>		
		2	
		4	
		8	
		16	the number elements per row (before advancing vert stride)
<i>HzStride</i>	-	0	
	>		
		1	
		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

<i>Names</i>	->	<i>AddrRegName</i>	address register
		<i>acc0</i>	
		...	
		<i>acc11</i>	up to <i>acc1</i> previously and <i>acc9</i>
		<i>FlagRegName</i>	f0 or f1
		<i>fc0</i>	
		...	
		<i>fc4</i>	flow control registers
		<i>ce</i>	channel enable register
		<i>cr0</i>	
		<i>ip</i>	instruction pointer
		<i>n0</i>	
		<i>n1</i>	
		<i>msg0</i>	
		...	
		<i>msg31</i>	
		<i>null</i>	the null register

		r0	general registers
		...	
		r255	
		sr0	
		sr1	
		tdr	
		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 ...			
<i>SubReg</i>	->	<i>IntLit</i>	

A reference to a flag register. Examples: f0.0 or f1.1			
FlagRegRef	->	<i>FlagRegName</i> (<i>. FlagSubReg</i>)?	
where			
<i>FlagSubReg</i>	->	0 1	

FlagRegName	->	f0 ... f3	

An address register reference; e.g. a0.3 (sometimes called the "index register")			
AddrRegRef	->	<i>AddrRegName</i> (<i>. AddrSubReg</i>)?	
where			
<i>AddrSubReg</i>	->	0 ... 15	

AddrRegName	->	a0	

Destination types are the allowable types on registers			
DstType	->	<i>RegType</i>	

Source types are the same as the destination types with the inclusion of the packed vector types.			
SrcImmType	->	<i>ImmType</i>	
SrcRegType	->	<i>RegType</i>	

Register and immediate operand types. RegTypes includes non-vector immediates.			
--	--	--	--

RegType	->	:b	8b signed integer
		:ub	8b unsigned integer
		: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	->	<i>IntLit</i>	integer literal
		<i>FtLit</i>	floating point literal
		<i>Ident</i>	reference to a symbolic constant
		<i>UnOp ConstExpr</i>	unary constant expressions (e.g. $\sim((1\text{NUM_BITS})-1)$)
		<i>ConstExpr BinOp ConstExpr</i>	binary constant expressions (1+2 or 1BIT_INDEX)
		<i>(ConstExpr)</i>	constant grouping expression (1 + 2)
		<i>LabelRef</i>	a reference to a label definition

Unary operators			
UnOp	->	~	one's complement
		-	two's complement

Binary operators (in precedence order)			
BinOp	->	*	multiplicative operators
		/	
		%	
		+	additive operators
		-	
		<<	bit-shift operators
		>>	
		>>>	
		&	bitwise (logical) operators
		^	

InstOptions	->	{ (InstOpt (, InstOpt)*)? }
where		
<i>InstOpt</i>	->	AccWrEn
		Atomic
		Breakpoint
		Compacted
		EOT
		DeplInfo

DeplInfo	->	RegDist SBIDAlloc SBIDWaitSrc SBIDWaitDst	
where			
RegDist	->	@IntLit	Minimum distance on any 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)
SBIDWaitSrc	->	$\$IntLit.src$	stall until sources read
SBIDWaitDst	->	$\$IntLit.dst$	stall until writeback complete

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
		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

This page defines the extended load/store syntax that an assembler can use to represent send messages. In all cases, these forms reduce to the canonical send format with raw hex descriptors. In other words, these instructions are *pseudo-instructions* and encode exactly as the old send messages.

Normally send messages are something like the following example:

```
sends (8) r12 r13 r14 0xC 0x410214
```

This encodes a regular ISA instruction with the opcode for `sends`. However, the two hex fields given as the final operands of the syntax represent the send message instruction type that the load/store units (sampler/HDC/etc) decode. Hence, a `send` is an instruction within an instruction of sorts. However, this makes it difficult to determine exactly what message represents. This syntax attempts to resolve that with load and store pseudo-instructions.

This encodes a regular ISA instruction with the opcode for `sends`. However, the two hex fields given as the final operands of the syntax represent the send message instruction type that the load/store units (HDC, etc) decode. Hence, a `send` is an instruction within an instruction of sorts. However, this makes it difficult to determine exactly what message represents. This syntax attempts to resolve that with load and store pseudo-instructions.

The field encoded within those two descriptors are described elsewhere.

Syntax

Syntax is broken down into several categories such as loads, stores, atomics, and perhaps other operations.

Loads

Loads are generally of the following form.

```
ld.OP[.ARGS] (ExecSize|ChOff) LdDst LdSrc LdStInstOpts
```

where *OP[.ARGS]* are given in the table *Mnemonic* column below.

Stores

Stores have the following form.

```
st.OP[.ARGS] (ExecSize|ChOff) StDst StSrc LdStInstOpts
```

Operand Syntax

LdDst	- >	RegRange	
LdSrc	- >	AddrOperand	
StDst	- >	AddrOperand	
StSrc	- >	RegRange	
AddrOperand	- >	AddrModel [RegRange]	e.g. a64[r12-13]
		AddrModel [RegRange + ConstIntExpr]	This for messages that permit a constant offset
AddrModel	- >	a32	Stateless A32 Non-Coherent (bti 0xFD)
		a32c	Stateless A32 Coherent (bti 0xFF)
		a32o	Stateless A32 Surface State Offset Model (bti 0xFC)
		a64	Stateless A64 Non-Coherent (bti 0xFD)
		a64c	Stateless A64 Coherent (bti 0xFF)
		a64o	Stateless A64 Surface State Offset Model (bti 0xFC)
		surf [ConstIntExpr]	Binding Table Index
		scratch [ConstIntExpr]	Indicates a Scratch Access. The index is in HWords / register units (32 bytes).
RegRange	- >	SingleReg-RegNum	Indicates an inclusive register range: e.g. r12-13
		SingleReg	Shorthand for a single register range: e.g. r13 is the same as r13-13
SingleReg	- >	r0 r1 ...	Maximum value can vary per platform. E.g. r0 up to r127
RegNum	- >	0 1 ...	An integer within the range of registers. Must be greater than or equal to the base register in the range. The maximum register varies per platform and configuration.

Examples

Here are some example messages.

```
// A SIMD16 (16|M0) byte gathering read (ld.ga8)
// The 16 addresses are read from r20-21 (wlen=2) and access surface 0 (surf[0]).
// Each address looks up four bytes (.x4).
// Output is stored in r10-17 (rlen=8)
ld.sc8.x4 (16|M0) r10-17 surf[0][r20-21]
// A Dword gathering read (ld.ga32) with one DW per address (.x1)
// The read is SIMD8 (8|M0) in the A64 stateless model (a64). Addresses are in r20-21.
ld.sc32.x1 (8|M0) r10-17 a64[r20-21]
```

Supported Messages

Some basic points apply for tools.

1. The table is incomplete and some message types are omitted from this extended syntax. In these cases, assemblers may fall back to the canonical send syntax with raw hex descriptors.
2. The table below takes some liberties simplifying the encoding given by the official BXML structures. For instance, some of these structures show an *Invalidate After Read* bit (which is always zero). The formats below might simplify this to just a reserved field with 0. For a second example consider the message type field, which is sometimes bits [18:14] and sometimes [17:14] with a reserved or fixed bit value at bit [18] as a separate field. In that case, we present the table with [18:14] = {[18][17:14]}: a logical fusing of the two fields into one so that more message types can use [18:14] for the same purpose.
3. In cases where a send instruction uses a register descriptor (a0), we must fall back to the original send syntax.
4. In the table below the descriptor fields for message length (mlen [28:25]) and writeback length (rlen [24:20]) are omitted (represented by ... in a middle column). These values are inferred via the register operand ranges.
5. Reserved fields (R) will typically be 0, but may be ignored.

Vector Messages

Mnemonic	Name	BXML Structure	SFI																								
			D	1	0	29	..	19	8	7	6	5	4	13	12	1	0	9	8	7	6	5	4	3	2	1	0
ld.xga8.DE	A64 Byte Gathering Read	MSD1R_A64_BS	DC1	R	P	0x0	..	0x0	0x10		R	SM2	DE	0x0													A64
st.xsc8.DE	A64 Byte Scattering Write	MSD1W_A64_BS	DC1	R	P	0x0	..	0x0	0x1A		R	SM2	DE	0x0													A64
ld.xga32.DE	A64 DWord Gathering Read	MSD1R_A64_DWS	DC1	R	P	0x0	..	0x0	0x10		R	SM2	DE	0x1													A64
st.xsc32.DE	A64 DWord Scattering Write	MSD1W_A64_DWS	DC1	R	P	0x0	..	0x0	0x1A		R	SM2	DE	0x1													A64
ld.xga64.DE	A64 QWord Gathering Read	MSD1R_A64_QWS	DC1	R	P	0x0	..	0x0	0x10		R	SM2	DE	0x2													A64
st.xsc64.DE	A64 QWord Scattering Write	MSD1W_A64_QWS	DC1	R	P	0x0	..	0x0	0x1A		R	SM2	DE	0x2													A64
ld.ga8.DE	A32 Byte Gathering Read	MSD0R_BS	DC0	R	P	0x0	..	H	0x4		R		DE	R	SM2												BTS_SLM_A32
st.sc8.DE	A32 Byte Scattering Write	MSD0W_BS	DC0	R	P	0x0	..	H	0xC		R		DE	R	SM2												BTS_SLM_A32
ld.ga32.DE	A32 DWord Gathering Read	MSD0R_DWS	DC0	R	P	0x0	..	H	0x3		R		DE	0x1	SM2												BTS_SLM_A32
st.sc32.DE	A32 DWord Scattering Write	MSD0W_DWS	DC0	R	P	0x0	..	H	0xB		R		DE	0x	SM												BTS_SLM_A32

Block Messages

Mnemonic	Name	BXML Structure	3 3 2 .. 1 1 1 1 1 1 1 1 1 1																									
			SFID	1	0	9	.	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	
ld.xb1256.H <i>W64</i>	A64 HWord Block Read	MSD1R_A64_H WB	DC1	R			...	H	0x14					R	0x3	HW64	A64											
st.xb1256.H <i>W64</i>	A64 HWord Block Write	MSD1W_A64_H WB	DC1	R			...	H	0x15					R	0x3	HW64	A64											
ld.xb1128.O <i>Ws</i>	A64 OWord Block Read	MSD1R_A64_O WB	DC1	R			...	H	0x14					R	0x0	OWs	A64											
st.xb1128.O <i>Ws</i>	A64 OWord Block Write	MSD1W_A64_O WB	DC1	R			...	H	0x15					R	0x0	OWs	A64											
ld.xub1128.O <i>OWs</i>	A64 Unalign ed OWord Block Read	MSD1R_A64_O WUB	DC1	R			...	H	0x15					R	0x1	OWs	A64											
ld.b1128.O <i>Ws</i>	A32 OWord Block Read	MSDOR_OWB	DC0	R			...	H	0x0					R		OWs	BTS_SLM_A32											
st.b1128.O <i>Ws</i>	A32 OWord Block Write	MSDOW_OWB	DC0	R			...	H	0x8					R		OWs	BTS_SLM_A32											
ld.ab1128.O <i>Ws</i>	A32 Aligned OWord Block Read	MSDOR_OWAB	DC0	R			...	H	0x1					R		OWs	BTS_SLM_A32											
ld.cb1128.O <i>Ws</i>	A32 Constan t OWord Block Read	MSDCR_OWB	DCR O	R			...	H	0x0					R		OWs	BTS											
ld.cub1128.O <i>OWs</i>	A32 Constan t Unalign ed	MSDCR_OWUB	DCR O	R			...	H	0x1					R		OWs	BTS											

Mnemonic	Name	BXML Structure	3 3 2 .. 1 1 1 1 1 1 1 1 1 1																								
			SFID	1	0	9	.	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
	OWord Block Read																										
ld.sb1256.H <i>WSB</i>	A32 Scratch HWord Block Read	MSDOR_SB	DC0	R			...	H	0x10								HWSB										ScratchOffset
st.sb1256.H <i>WSB</i>	A32 Scratch HWord Block Write	MSDOW_SB	DC0	R			...	H	0x18								HWSB										ScratchOffset

Surface Messages

Mnemonic	Name	BXML Structure	SFI 3 3 .. 1 1 1 1 1 1 1 1 1 1																								
			D	1	0	29	.	19	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
ld.xus.CMask	A64 Untyped Surface Read	MSD1R_A64_US	DC1	R			...	0	0x11								SM3	CMask									A64
st.xus.CMaskUW	A64 Untyped Surface Write	MSD1W_A64_US	DC1	R			...	0	0x19								SM3	CMaskUW									A64
ld.us.CMask	A32 Untyped Surface Read	MSD1R_US	DC1	R			...	H	0x1								SM3	CMask									BTS_SLM_A32
st.us.CMaskUW	A32 Untyped Surface Write	MSD1W_US	DC1	R			...	H	0x9								SM3	CMaskUW									BTS_SLM_A32
ld.ts.SG3.CMask	A32 Typed Surface Read	MSD1R_TS	DC1	R			...	H	0x5								SG3	CMask									BTS
st.ts.SG3.CMaskUW	A32 Typed Surface Write	MSD1W_TS	DC1	R			...	H	0xD								SG3	CMaskUW									BTS

Mnemonic	Name	BXML Structure	SFI 3 3 .. 1 1 1 1 1 1 1 1 1 1 1 1 1 1																								
			D	1	0	29	.	19	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
	Surface Write																										
ld.hus.CMask	A16 Untyped Surface Read	MSD1R_A16_US	DC 1	R		0x1	..	H	0x1							SM3	CMask										BTS_SLM_A32
ld.huss.CMask	A16 Untyped Surface Read	MSD1RS_A16_US	DC 1	R		0x1	..	H	0x8							SM3	CMask										BTS_SLM_A32
st.hus.CMaskUW	A16 Untyped Surface Write	MSD1W_A16_US	DC 1	R		0x1	..	H	0x9							SM3	CMaskUW										BTS_SLM_A32
ld.hts.SG3.CMask	A16 Typed Surface Read	MSD1R_A16_TS	DC 1	R		0x1	..	H	0x5							SG3	CMask										BTS
st.hts.SG3.CMaskUW	A16 Typed Surface Write	MSD1W_A16_TS	DC 1	R		0x1	..	H	0xD							SG3	CMaskUW										BTS
ld.htss.SG3.CMask	A16 Typed Surface Read	MSD1RS_A16_TS	DC 1	R		0x1	..	H	0xE							SG3	CMask										BTS

For the read message, indicates that which channels are read from the surface and included in the writeback message. For the write message, indicates which channels are included in the message payload and written to the surface.

Syntax Mapping This field is determined via an argument [*CMask*] to the load/store.
 E.g. `ld...CMask (8) data ...[addr]`
 E.g. `st...CMask (8) [addr] data`

Symbol	Value	Desc.
rgba	0x0	
gba	0x1	
rba	0x2	
ba	0x3	
rga	0x4	
ga	0x5	
ra	0x6	
a	0x7	
rgb	0x8	
gb	0x9	
rb	0xA	
b	0xB	
rg	0xC	
g	0xD	
r	0xE	

CMaskUW

Untyped Write Channel Mask Message Descriptor Control Field

For the read message, indicates that which channels are read from the surface and included in the writeback message. For the write message, indicates which channels are included in the message payload and written to the surface.

Syntax Mapping This field is determined via an argument [*CMaskUW*] to the load/store.
 E.g. `ld...CMaskUW (8) data ...[addr]`
 E.g. `st...CMaskUW (8) [addr] data`

Symbol	Value	Desc.
rgba	0x0	
rgb	0x8	
rg	0xC	
r	0xE	

DE

Data Elements

The number of elements per channel (address). Values are returned in SOA form. E.g. x2 will return all the X's for each channel followed by the Y's. A SIMD16 message will consume a pair of registers for each element. Byte values are aligned as the low byte of each Dword (so only 8 per register).

Syntax Mapping
 This field is determined via an argument *[DE]* to the load/store.
 E.g. `ld...DE (8) data ...[addr]`
 E.g. `st...DE (8) [addr] data`

Symbol	Value	Desc.
x1	0x0	1 element per address
x2	0x1	2 elements per address
x4	0x2	4 elements per address
x8	0x3	8 elements per address

H

Header Optional

Indicates that a header is optional

Syntax Mapping
 The header presence is implied via the number of address registers.
 E.g. `ld... (8) ... a32[r12-13]`
 Since the message is A32 SIMD8 (1 GRF for addresses), the first of the two address GRFs is assumed to be a header.
 E.g. `ld... (8) ... a32[r12]`
 Since the message is A32 SIMD8 (1 GRF for addresses), the first of the two address GRFs is assumed to be a header.
 E.g. `ld... (8) ... a32[r12,r24]`
 Since the message is A32 SIMD8 (1 GRF for addresses), and there are two GRF addresses given (as split arguments), the header is inferred to be included.
 E.g. `ld... (16) ... a64[r12-16]`
 Since the message is A64 SIMD16 it has 4 GRFs as addresses. Hence given 5 address registers, a header is included.

Symbol	Value	Desc.
N/A	0x0	header omitted
N/A	0x1	header included

HW64

Data Blocks Hex Words

The number of 256b data blocks to fetch for the given address (for A64 HWord messages).

Syntax Mapping
 This field is determined via an argument *[HW64]* to the load/store.
 E.g. `ld...HW64 (8) data ...[addr]`
 E.g. `st...HW64 (8) [addr] data`

Symbol	Value	Desc.
x1	0x1	1 HWord
x2	0x2	2 HWords

×4	0x3	4 HWords
×8	0x4	8 HWords

HWSB

Data Blocks Hex Words

The number of 256b data blocks to fetch for the given address (for Non-A64 messages).

Syntax Mapping	This field is determined via an argument [<i>HWSB</i>] to the load/store. E.g. <code>ld...HWSB (8) data ...[addr]</code> E.g. <code>st...HWSB (8) [addr] data</code>
----------------	--

Symbol	Value	Desc.
×1	0x0	1 HWord (a full GRF)
×2	0x1	2 HWords
×4	0x2	4 HWords
×8	0x3	8 HWords

OWs

Data Blocks Oct Words

The number of 128b data blocks to fetch for the given address.

Syntax Mapping	This field is determined via an argument [<i>OWs</i>] to the load/store. E.g. <code>ld...OWs (8) data ...[addr]</code> E.g. <code>st...OWs (8) [addr] data</code>
----------------	---

Symbol	Value	Desc.
×1L	0x0	1 OWord accessed in the low half of the GRF
×1H	0x1	1 OWord accessed in the high half of the GRF
×2	0x2	2 OWords (1 full GRF)
×4	0x3	4 OWords (2 full GRF)
×8	0x4	8 OWords (4 full GRF)

PD

Packed Data

A value of one indicates that the data is packed.

Syntax Mapping	This field is set if the instruction option PackedData. E.g. <code>ld... (8) dst ...[src] {PackedData}</code>
----------------	--

Symbol	Value	Desc.
N/A	0x0	unpacked data
N/A	0x1	packed data

SG3

Slot Group 3		
Syntax Mapping	This field is determined via an argument <i>[SG3]</i> to the load/store. E.g. <code>ld...SG3 (8) data ...[addr]</code> E.g. <code>st...SG3 (8) [addr] data</code>	
Symbol	Value	Desc.
sg4x2	0x0	
sg8l	0x1	
sg8h	0x2	
SM2		
SIMD Mode 2		
this value is inferred from the instruction's <i>ExecSize</i>		
Syntax Mapping	This field is inferred via the instruction's execution size. Note: sizes smaller than SIMD8 are rounded up to SIMD8. E.g. <code>ld... (4) ...</code> will choose a SIMD8 message size. The instruction's <i>ExecSize</i> will still be SIMD4.	
Symbol	Value	Desc.
SIMD8	0x0	8 addresses are sent out
SIMD16	0x1	16 addresses are sent out
SM3		
SIMD Mode 3		
this value is inferred from the instruction's <i>ExecSize</i>		
Syntax Mapping	This field is inferred via the instruction's execution size. Note: sizes smaller than SIMD8 are rounded up to SIMD8. E.g. <code>ld... (4) ...</code> will choose a SIMD8 message size. The instruction's <i>ExecSize</i> will still be SIMD4.	
Symbol	Value	Desc.
SIMD16	0x1	16 channels (addresses)
SIMD8	0x2	8 channels (addresses)
ScratchOffset		
Scratch Offset		
In 32B chunks (i.e. the encoded value is scaled by 32; the syntax emits the value in bytes).		
Syntax Mapping	The address model is inferred via a lexeme preceding the address operand. E.g. <code>ld... (8) ... scratch[.] [...]</code> E.g. <code>st... (8) a32 [...]</code>	
Symbol	Value	Desc.

scratch[...][...]	0x0-FFF	scratch offset
-------------------	---------	----------------

Split-Sends and Conditional Sends

A final minor complication to be dealt with involves how we map to specific EU ISA send ops, since there are several in that family: send, sendc, sends, sendsc. If the underlying send instruction desired is a unary send, we affix a p or c to the ld/st operand. Hence, the following mapping applies.

Pseudo-Instruction	Encodes As
ld	send (send is binary just as in prior sends)
ldc	sendc
ldp (p for packed)	Unsupported
ldcp	Unsupported

The intention of the mapping above is to make the common and future cases the most concise.

Operand Mapping

This section describes how send operands are mapped to load/store syntax.

Load Pseudo-Instructions

Send instructions that representing load operations come in several forms. The figure below illustrates some examples below. Each box represents a GRF. For the vector messages, assume a 16 x 32-bit addresses are sent out (without loss of generality).

Scalar (e.g. Block Messages)		Vector with a Header (e.g. Gathering or Surface Read messages)		Vector (Headerless)	
src0	Header	src0	Header	src0	Addr0
		src1	Addr0	src0+1	Addr1
		src1+1	Addr0		

Block Messages

Load Syntax		Send Encoding	Notes
ld.bl256.x2 (1) dst a64[hdr]	<====>	sends (1) dst hdr null ...	
ldc.bl256.x2 (1) dst a64[hdr]	<====>	sendsc (1) dst hdr null ...	
ldp.bl256.x2 (1) dst a64[hdr]	<====>	send (1) dst hdr ...	(Only if unary send supported)

For block messages, only a header is sent (*mLen* is one with the header-present bit set.) The header/address is held in *src0* and *src1* is left null (if applicable). Note that the header (*hdr*) is the address in block messages.

Vector Messages with a Header

Load Syntax		Send Encoding	Notes
ld.ga32.x1 (16) dst slm[hdr,addr]	<====>	sends (16) dst hdr addr ...	
ldp.ga32.x1 (16) dst slm[src0]	<====>	send (16) dst (hdr:addr)	(Only if unary send supported)

For vector messages with a header `src0` holds the header and `src1` holds the address payloads (the first example above). The second illustrates that both the header and addresses can be concatenated into one operand (if they are in successive GRFs).

Headerless Vector Messages

Load Syntax		Send Encoding	Notes
ld.ga8.x4 (16) dst surf[1][addr]	<====>	sends (8) dst addr null ...	
ldp.ga8.x4 (16) dst surf[1][addr]	<====>	send (8) dst addr ...	(Only if unary send supported)

For vector messages without a header `src0` holds all addresses and `src1` is `null` (if applicable).

Store Pseudo-Instructions

Store instructions are encoded as send instructions. Since there is no return value ($rLen = 0$), the `send`'s encoded destination parameter is `null`. For the example below, assume the vector messages have 16 x 32b addresses to be sent out and 64 bytes of data are being sent out (2 GRFs).

Scalar (e.g. Block Messages)		Vector with a Header (e.g. Scattering Write or Surface Write messages)		Vector (Headerless)	
src0	Header	src0	Header	src0	Addr0
src1	Data0	src0+1	Addr0	src0+1	Addr1
src1+1	Data1	src0+2	Addr0	src1	Data0
		src1	Data0	src1+1	Data1
		src1+1	Data1		

Block Messages

Store Syntax		Send Encoding	Notes
st.bl256.x2 (1) a64[hdr] data	<=====>	sends (8) null hdr data ...	
stp.bl256.x2 (1) a64[(hdr:data)] data	<=====>	send (1) dst src0 ...	Src0 holds both the header (address) and the data registers. We replicate the data portion in the store instruction's source for readability, but it is not explicitly encoded into the send.
For block messages src0 holds the header/address and src1 holds the data.			

Vector Messages with a Header

Store Syntax		Send Encoding	Notes
<code>st.sc8.x4 (8) slm[(hdr:addr)] data</code>	<code><====></code>	sends (8) null (hdr:addr) data	The header and address are packed into <code>src0</code> (in successive GRFs).
<code>stp.sc8.x4 (8) slm[(hdr:addr:data)] data</code>	<code><====></code>	send (16) null (hdr:addr:data) ...	<code>src0</code> holds both header and address

For vector messages with a header `src0` holds the header and addresses and `src1` holds the data payload. In the packed version (where supported) everything can be combined into one operand if placed in successive registers of GRF. We present the data part as an operand in syntax, but it does not get encoded explicitly in the send (*mlen* is set accordingly).

Headerless Vector Messages

Load Syntax		Send Encoding	Notes
<code>st.sc8.x4 (8) slm[addr] data</code>	<code><====></code>	sends (8) null addr data ...	

For vector messages without a header `src0` holds the addresses and `src1` holds the data.

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, $\text{floor}(1.3f) = 1.0f$ and $\text{floor}(-1.3f) = -2.0f$.

```
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
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
brc	Branch Converging	EUISA
brd	Branch Diverging	EUISA

Symbol	Name	Source
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
dp4a	Dot Product 4 Accumulate	EUISA
else	Else	EUISA
endif	End If	EUISA
math	Extended Math Function <ul style="list-style-type: none"> • INV - Inverse • LOG - Logarithm • EXP - Exponent • SQRT - Square Root • RSQ - Reciprocal Square Root • POW - Power Function • SIN - SINE • COS - COSINE • INT DIV - Integer Divide • INVM/RSQRTM 	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
subb	Integer Subtraction with Borrow	EUISA
join	Join	EUISA
jmpj	Jump Indexed	EUISA
lzd	Leading Zero Detection	EUISA
and	Logic And	EUISA
not	Logic Not	EUISA

Symbol	Name	Source
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 <ul style="list-style-type: none"> ▪ 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
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

<i>src0</i>	-inf	-finite	-denorm	-0	+0	+denorm	+finite	+inf	NaN
<i>dst</i>	-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

<i>src0</i>	-fmax	-finite	-denorm	-0	+0	+denorm	+finite	+fmax	***
<i>dst</i>	-fmax	-finite	-0	-0	+0	+0	**	+fmax	
Notes:									
**	Result may be {+finite, +0}.								
***	Result is undefined if <i>src0</i> 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 the 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 <i>src0</i> 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 `ceil()` 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}.																												
^	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="10">Note</td> </tr> <tr> <td colspan="10">{+1, +0} depending on the Single Precision Denorm Mode.</td> </tr> </table>									Note										{+1, +0} depending on the Single Precision Denorm Mode.									
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 <i>src0</i> 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 <i>src0</i> is {-inf, +inf, NaN}.								

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	SQRT
5h	RSQ
6h	SIN
7h	COS
8h	Reserved
Bh	INT DIV - return quotient and remainder
Ch	INT DIV - return quotient only
Dh	INT DIV - return remainder

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 $\text{Log}_2(\text{abs}(\text{src0}))$

EXP - Exponent

Precision: $\leq 3\text{ULP}$

SQRT - Square Root

Precision: $\leq 1\text{ULP}$

Notes: In ALT mode SQRT is computed as $\text{SQRT}(\text{abs}(\text{src0}))$



RSQ - Reciprocal Square Root

Precision: $\leq 3\text{ULP}$

Notes: In ALT mode RSQ is computed as $\text{RSQ}(\text{abs}(\text{src0}))$

SIN - SINE

Precision: Absolute error ≤ 0.0008 for the range of $\pm 32767 \cdot \pi$

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 $\pm 32767 \cdot \pi$

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.

INT DIV - Integer Divide

Precision: 32-bit integer

For signed inputs, INT DIV behavior is illustrated by the table below:

Inputs:	Numerator	+	+	-	-
	Denominator	+	-	+	-
Outputs:	Quotient	+	-	-	+
	Remainder	+	+	-	-

INVM/RSQRTM

These are special opcodes. These two special opcodes INVM and RSQRTM provide the initial approximation for a macro that finally produces an IEEE compliant FDIV and SQRT respectively. They perform the same operation as INV and RSQRT. However, the outputs of these opcodes have higher precision, i.e., 34 bits for single precision and 66 bits for double precision. 32 bits and 64 bits, for single and double precision respectively are stored in the GRF and the remaining bits are stored in the special accumulators acc2-acc9. These operations provide the initial approximation for inverse and sqrt operations. The final IEEE compliant values are derived through a series of special fma operations. The pseudo code for these macro operations is defined below.

Early Out (math.eo.f0.0) is implicitly set in the instruction when these two special opcodes INVM and RSQRTM are used.

Programming Notes: The GRF registers and accumulators used in the macros below are merely examples. There are no restrictions on using these in any specific order as long as the GRF register and the accumulator are paired correctly in writes and reads. Since early out handling requires flow control to channels that do not require compute through the macro, Single Program Flow must never be set for a macro operation.

Macro for Single Precision IEEE Compliant fdiv

[Note: Constants are in quotes.]

```

Set Rounding Mode in CR to RNE
GRF are initialized: r2 = 0, r6 = a, r4 = b, r5 = 1
The default data type for the macro is :f

math.eo.f0.0 (8) r8.acc2 r6.noacc r4.noacc 0xE
(-f0.0) if
madm (8) r9.acc3 r2.noacc r6.noacc r8.acc2 // Step(1), q0=a*y0
madm (8) r10.acc4 r5.noacc -r4.noacc r8.acc2 // Step(2), e0=(1-b*y0)
madm (8) r11.acc5 r8.acc2 r10.acc4 r8.acc2 // Step(3), y1=y0+e0*y0
madm (8) r11.acc6 r6.noacc -r4.noacc r9.acc3 // Step(4), r0=a-b*q0
madm (8) r9.acc7 r9.acc3 r11.acc6 r11.acc5 // Step(5), q1=q0+r0*y1
madm (8) r6.acc8 r6.noacc -r4.noacc r9.acc7 // Step(6), r1=a-b*q1

Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL

madm (8) r8.noacc r9.acc7 r6.acc8 r11.acc5 // Step(7), q=q1+r1*y1
endif

```

Macro for Double Precision IEEE Compliant fdiv

```

Set Rounding Mode in CR to RNE
GRF are initialized: r0 = 0, r6 = a, r7 = b, r1 = 1
The default data type for the macro is :df

math.eo.f0.0 (4) r8.acc2 r6.noacc r7.noacc 0xE
(-f0.0) if
madm (4) r9.acc3 r0.noacc r6.noacc r8.acc2 // Step(1), q0=a*y0
madm (4) r10.acc4 r1.noacc -r7.noacc r8.acc2 // Step(2), e0=(1-b*y0)
madm (4) r11.acc5 r6.noacc -r7.noacc r9.acc3 // Step(3), r0=a-b*q0
madm (4) r12.acc6 r8.acc2 r10.acc4 r8.acc2 // Step(4), y1=y0+e0*y0
madm (4) r13.acc7 r1.noacc -r7.noacc r12.acc6 // Step(5), e1=(1-b*y1)
madm (4) r8.acc8 r8.acc2 r10.acc4 r12.acc6 // Step(6), y2=y0+e0*y1
madm (4) r9.acc9 r9.acc3 r11.acc5 r12.acc6 // Step(7), q1=q0+r0*y1
madm (4) r12.acc2 r12.acc6 r8.acc8 r13.acc7 // Step(8), y3=y1+e1*y2
madm (4) r11.acc3 r6.noacc -r7.noacc r9.acc9 // Step(9), r1=a-b*q1

```



```
Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL
```

```
madm (4) r8.noacc r9.acc9 r11.acc3 r12.acc2 // Step(10), q=q1+r1*y3
endif
```

Macro for Single Precision IEEE Compliant sqrt

```
Set Rounding Mode in CR to RNE
GRF are initialized: r0 = 0, r8 = 1/2, r6 = a
The default datatype for the macro is :f
```

```
math.eo.f0.0 (8) r7.acc2 r6.noacc null 0xF
(-f0.0) if
madm (8) r9.acc3 r0.noacc r8.noacc r7.acc2 // Step(1), H0=1/2*y0
madm (8) r11.acc4 r0.noacc r6.noacc r7.acc2 // Step(2), S0=a*y0
madm (8) r10.acc5 r8.noacc -r11.acc4 r9.acc3 // Step(3), d0=1/2-S0*H0
madm (8) r9.acc6 r9.acc3 r10.acc5 r9.acc3 // Step(4), H1=H0+d0*H0
madm (8) r7.acc7 r11.acc4 r10.acc5 r11.acc4 // Step(5), S1=S0+d0*S0
madm (8) r10.acc8 r6.noacc -r7.acc7 r7.acc7 // Step(6), e0=a-S1*S1
```

```
Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL
```

```
madm (8) r7.noacc r7.acc7 r9.acc6 r10.acc8 // Step(7), S=S1+e0*H1
endif
```

Macro for Double Precision IEEE Compliant sqrt

```
Set Rounding Mode in CR to RNE
GRF are initialized: r0 = 0, r8 = 1/2, r6 = a, r1 = 1
The default datatype for the macro is :df
```

```
math.eo.f0.0 (4) r7.acc2 r6.noacc null 0xF
(-f0.0) if
madm (4) r9.acc3 r0.noacc r8.noacc r7.acc2 // Step(1), H0=1/2*y0
madm (4) r11.acc4 r0.noacc r6.noacc r7.acc2 // Step(2), S0=a*y0
madm (4) r10.acc5 r8.noacc -r11.acc4 r9.acc3 // Step(3), d=1/2-S0*H0
madm (4) r8.acc6 r1.noacc r8.noacc r1.noacc // Step(0'') 3/2=1+1/2
madm (4) r8.acc7 r1.noacc r8.acc6 r10.acc5 // Step(4), e=1+3/2*d
madm (4) r7.acc8 r0.noacc r10.acc5 r11.acc4 // Step(5), T0=d*S0
madm (4) r10.acc9 r0.noacc r10.acc5 r9.acc3 // Step(6), G0=d*H0
madm (4) r7.acc8 r11.acc4 r8.acc7 r7.acc8 // Step(7), S1=S0+e*T0
madm (4) r8.acc7 r9.acc3 r8.acc7 r10.acc9 // Step(8), H1=H0+e*G0
madm (4) r9.acc3 r6.noacc -r7.acc8 r7.acc8 // Step(9), d1=a-S1*S1
```

```
Change Rounding Mode in CR if required
Implicit Accumulator for destination is NULL
```

```
madm (4) r7.noacc r7.acc8 r9.acc3 r8.acc7 // Step(10), S=S1+d1*H1
endif
```

Restrictions for the Use of Macros

1. All operations are in Align 1 mode.
2. SubRegNum is re-purposed for implicit accumulator controls. Hence the default SubRegNum value is 0. (i.e. GRF aligned).
3. Macro Operations need to be used as special subroutines. Interleaving other operations is not recommended.
4. Single precision operations must be SIMD8 and double precision operations must be SIMD4.
5. The macro sequences should be used with the Rounding Mode in CR set to RNE. The Rounding Mode may be changed only for the last *madm* of the macro depending on the requirement.
6. When both srcs are NAN, FDIV produces denorminator NAN as output.

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 the syntax, gives examples, and provides fullsim extensions.

Fulsim Extensions

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 execution 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
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. Excution 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. Excution 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. 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 Than 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.

Execution Mask	
	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 its 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.
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 registers 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.
r19	Source 1.
0x10C	Extended Message Descriptor.
a0.0	Message Descriptor. Indirect. Value stored in ARF address register a0.0.



Control Flow Instructions

(W) jmp (1 M0) L1320	
(W)	Write Enable is ON.
jmp.	Jump Immediate.
(1 M0)	Execution Size: 1. Execution Mask Offset: 0.
L130	Label to jump to.
(W&~f0.1) jmp (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.

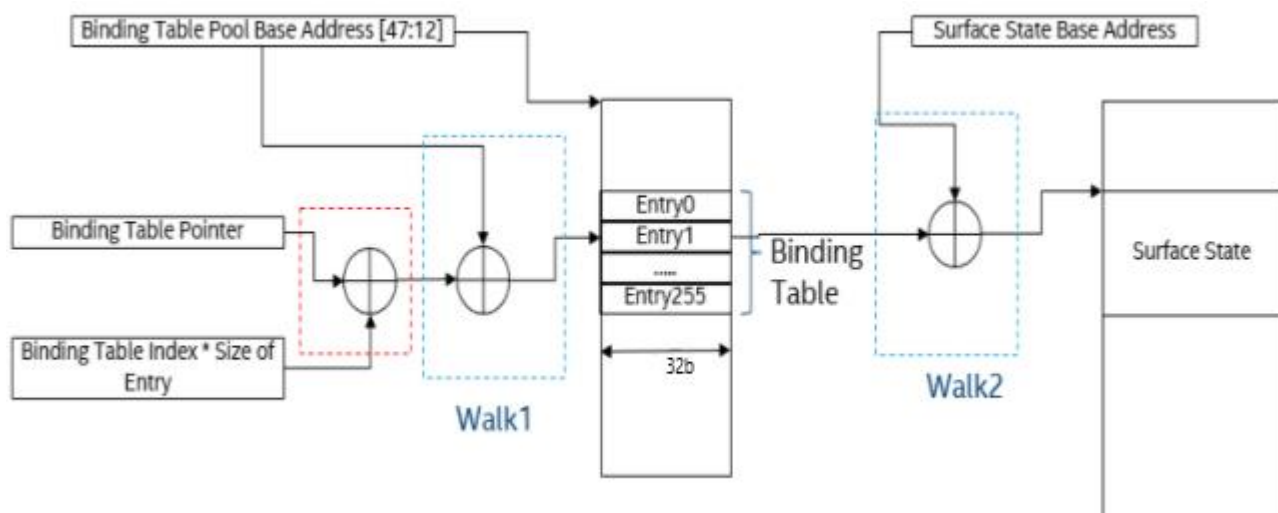
List of Shared Function Identifiers

Programming Note	
Context:	Shared Function ID
SFID_DP_DC1 is an extension of SFID_DP_DC0 to allow for more messages 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.	

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

Below is a table of the different entries and pointer offsets based on the Binding Table Pool and 18:8 Mode:

Render Pipe:

RCS Binding Table Pool Enable	HW Alignment 256B Mode	Binding Table Entry Format	Binding Table Pointer	Base Address
0	0	32 bit size/[31:6]	15:5	RCS Surface State Base Address
1	0	16 bit size/[21:6]	16:6	RCS Binding Table Pool Address
0	1	32 bit size/[31:6]	18:8	RCS Surface State Base Address
1	1	32 bit size/[31:6]	18:8	RCS Binding Table Pool Address

POSH Pipe(Note that the Pool Enable and 256B mode is shared with Render Pipe):

RCS Binding Table Pool Enable*	HW Alignment 256B Mode*	Binding Table Entry Format	Binding Table Pointer	Base Address
0	0	32 bit size/[31:6]	15:5	POCS Surface State Base Address
1	0	16 bit size/[21:6]	16:6	POCS Binding Table Pool Address
0	1	32 bit size/[31:6]	18:8	POCS Surface State Base Address
1	1	32 bit size/[31:6]	18:8	POCS Binding Table Pool Address

* SW must sync between the POCS and RCS prior to changing the binding table pool enable and 256B alignment mode.

Compute Pipe:

Binding Table Pool Enable	Binding Table Entry Format	Binding Table Pointer	Base Address
0	32 bit size/[31:6]	15:5	CCS Surface State Base Address
1	16 bit size/[21:6]	16:6	CCS 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**
- **HW 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 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 EU. The 3D Sampler can be used to assist the media sampler in specific operations such as video scaling.

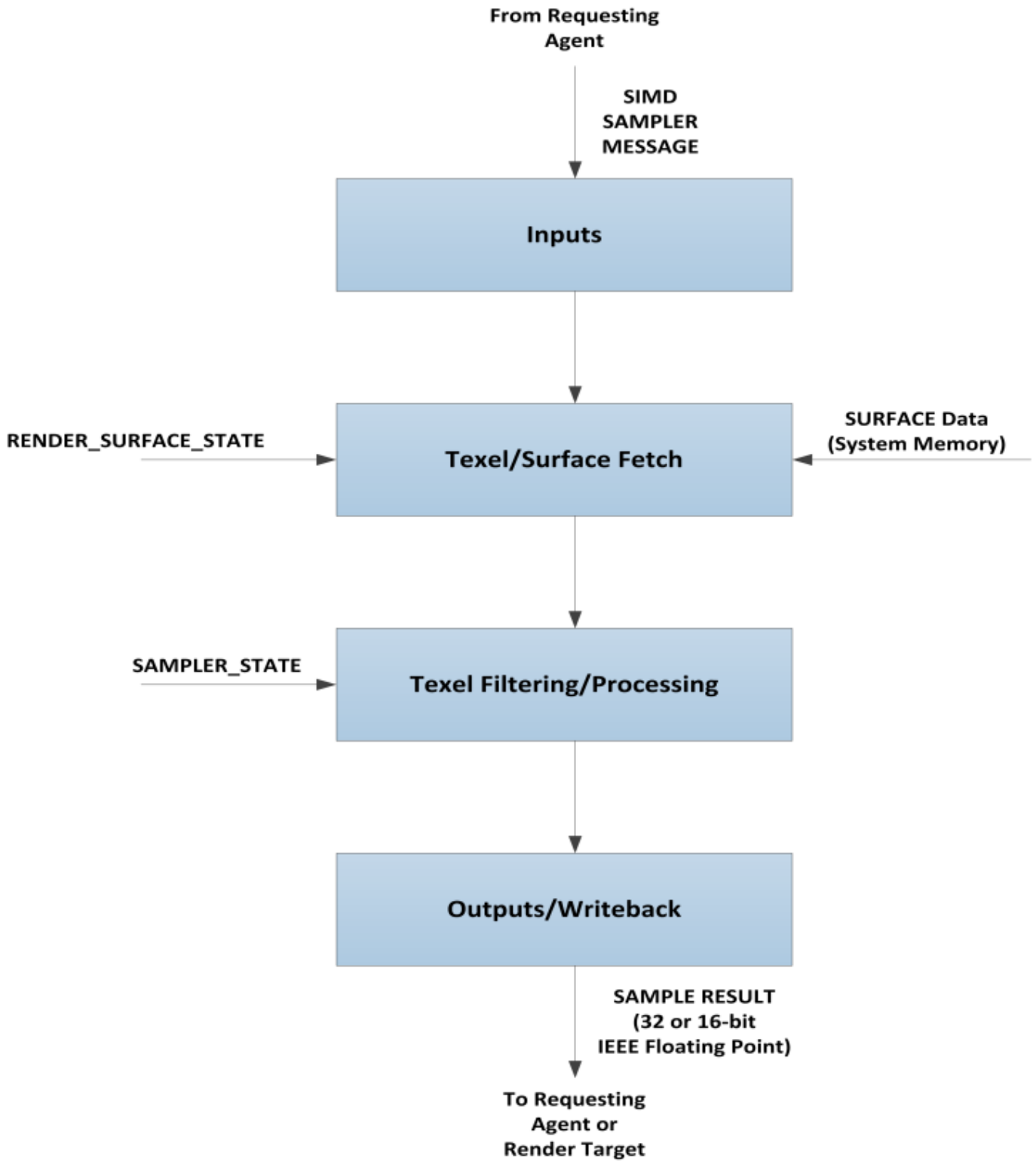
The following table summarizes the various subfunctions provided by the Sampling Engine. After the appropriate subfunctions are complete, the 4-component (reduced to fewer components in some cases) filtered texture value is provided to the EU to complete the *sample* instruction.

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.
Texture Palette Lookup	For streams which have "paletted" texture surface formats, this function uses the "index" values read from the texture map to look up texel color data from the texture palette.
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.
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 subslice which receives messages from other agents in the subslice, 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 kernel executing on the subslice 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.

Base format	New format	Channel select			
		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 no longer 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.

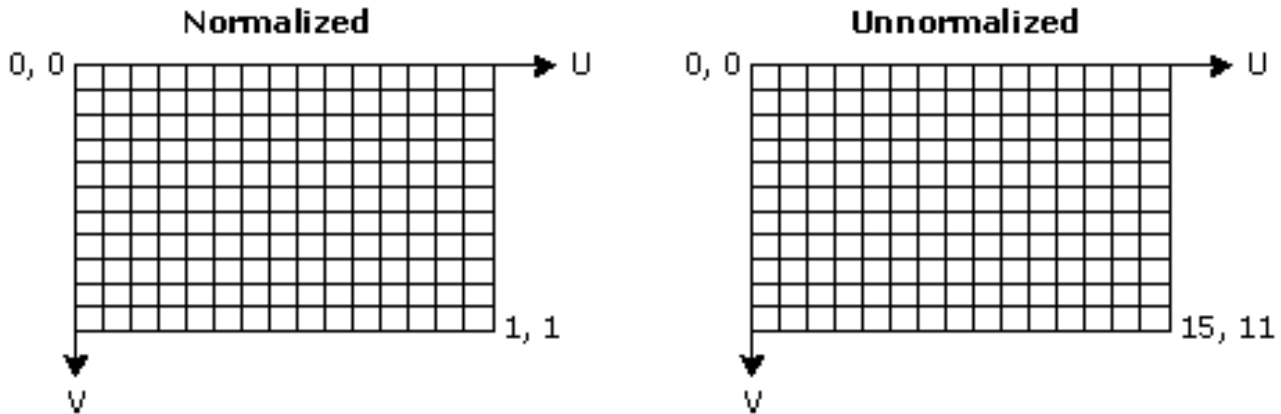
Texture Coordinate Normalization

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 located at the upper/left edge of the upper left texel of the base texture map.

Normalized vs. Unnormalized Texture Coordinates



B6877-01

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:



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

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

```

Pseudocode
<pre> 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) </pre>

MagMode = (LOD - Base <= 0)

Pseudocode
<pre> MagClampMipNone = LODClampMagnificationMode == MAG_CLAMP_MIPNONE </pre>

```

if ( (MagMode && MagClampMipNone) or MipFlt == None )

```

```

LOD = 0
LOD = min(LOD, ceil(AdjMaxLod))
LOD = max(LOD, floor(AdjMinLod))
else if ( MipFlt == Nearest )

```

Pseudocode
<pre> LOD = min(LOD, ceil(AdjMaxLod)) LOD = max(LOD, floor(AdjMinLod)) </pre>

```

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

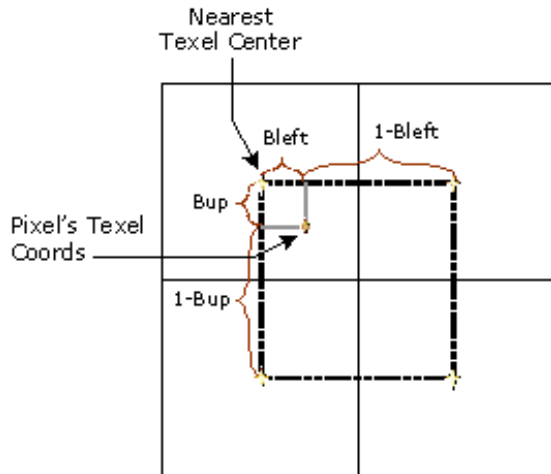
When **Corner Texel Mode** is enabled in the RENDER_SURFACE_STATE the definition of how the texel location is calculated is modified. Corner Texel Mode shifts the edge of the map to be coincident with the center of the edge texels. The 3D Sampler will handle this shift when sampling the texel.

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

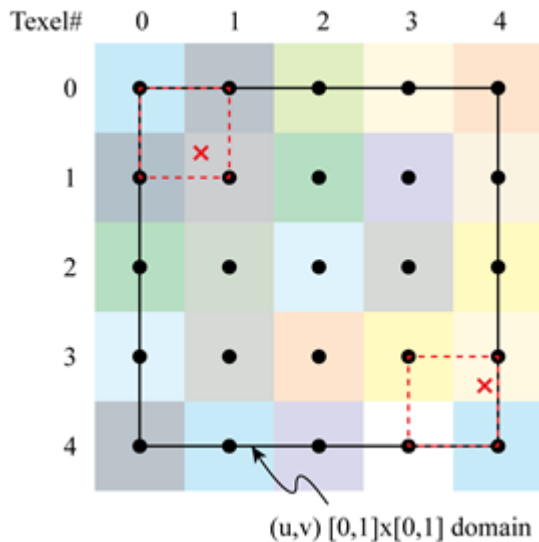


B 6879-01

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.



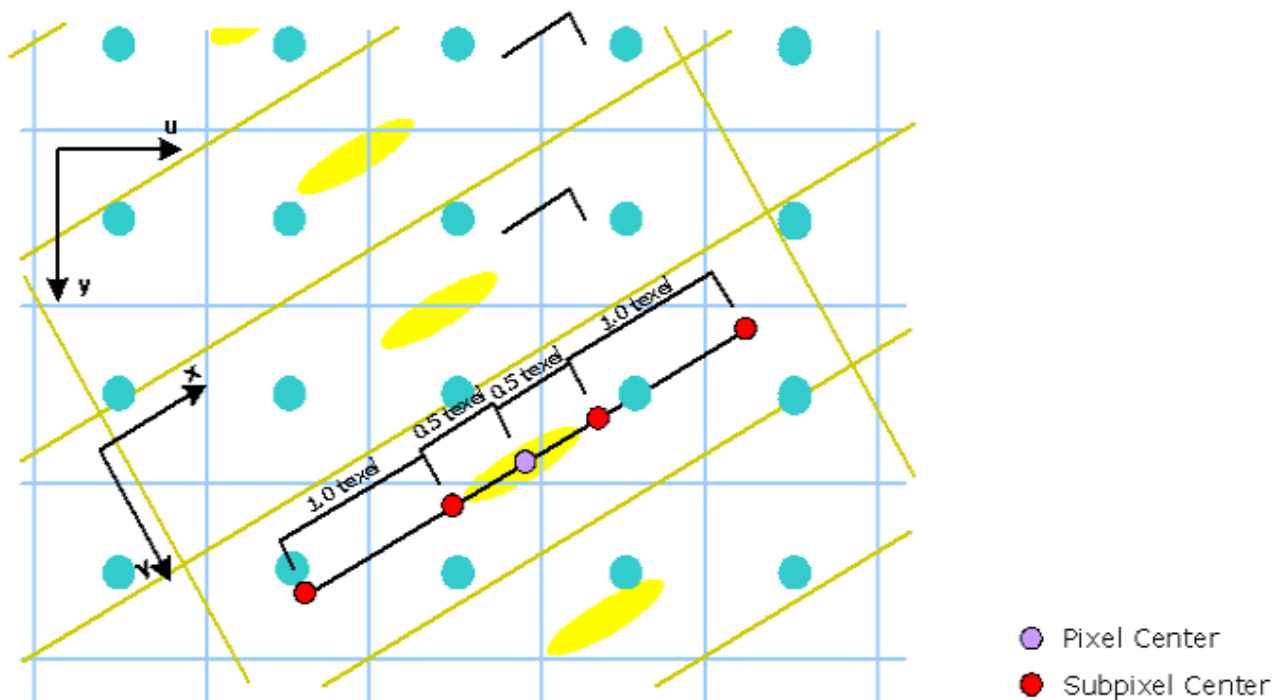
When doing linear filtering, the 3D sampler will treat a surface as being 1 texel less in each dimension when calculating texel location from the sample coordinates.

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.



B6880-01

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:

<i>MipFilter</i> 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.

<i>TC[X,Y,Z] Control</i>	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

<i>TC[X,Y,Z] Control</i>	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

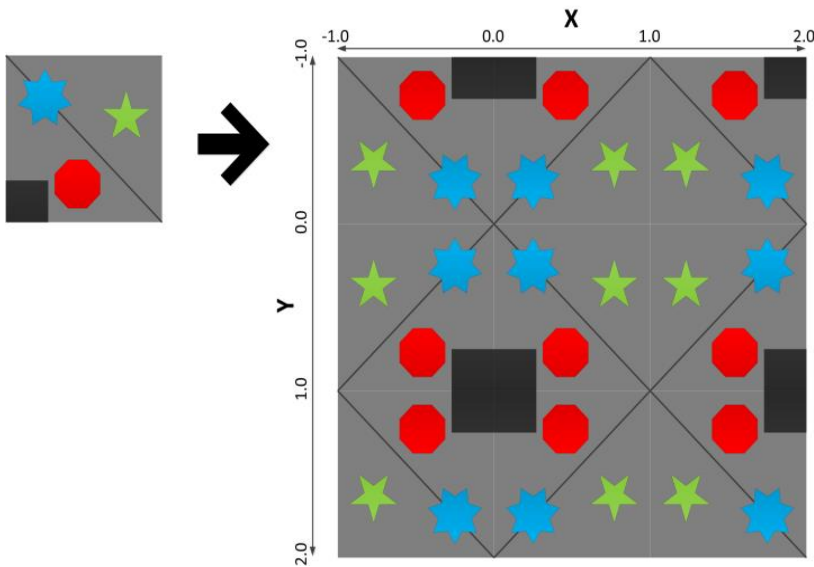
The number of map wraps on a given object is limited to 32. Going beyond this limit is legal but may result in artifacts due to insufficient internal precision, especially evident with larger surfaces. Precision loss starts at the subtexel level (slight color inaccuracies) and eventually reaches the 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.

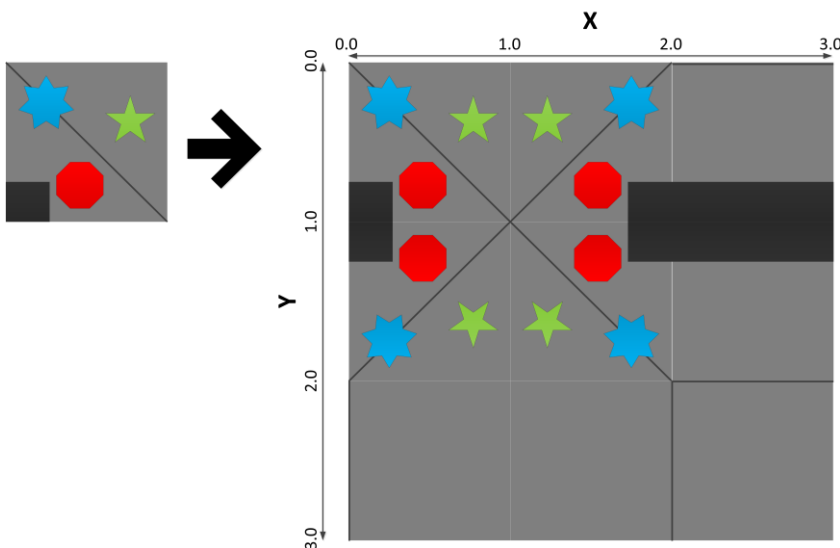
The example below shows how a simple 2D map with TEXCOORDMODE_MIRROR for both TCX and TCY is mapped.



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.

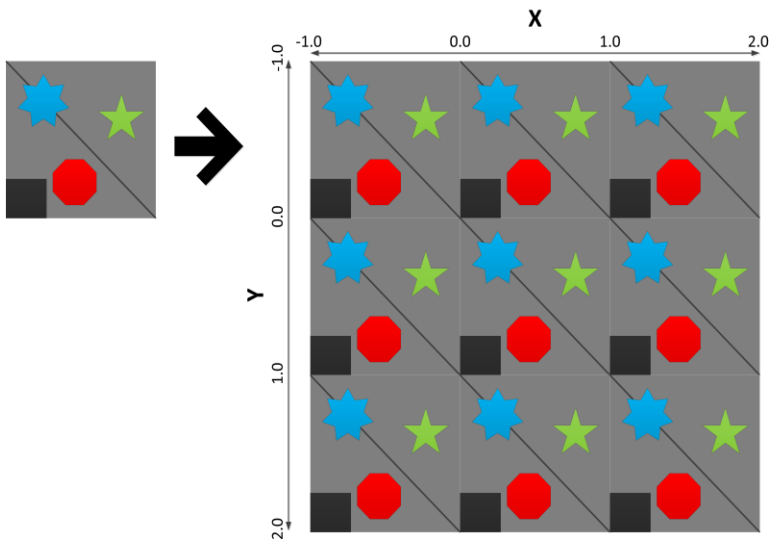
The example below shows how a simple 2D map with `TEXCOORDMODE_MIRROR_ONCE` for both `TCX` and `TCY` is mapped.



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.

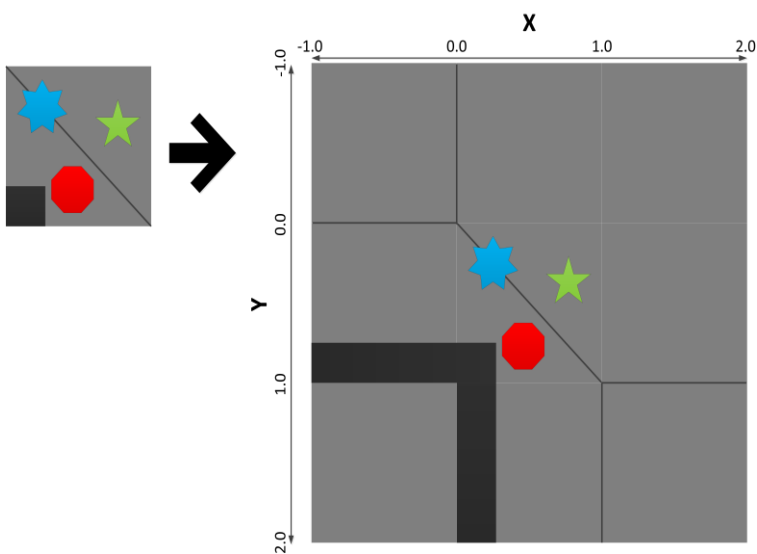


TEXCOORDMODE_CLAMP Mode

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.

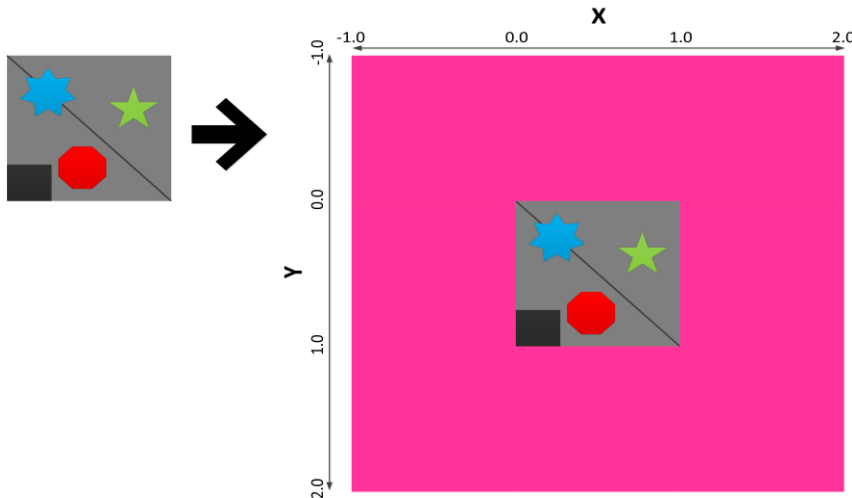
Texture Clamp Mode



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
MIRROR_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.

<i>ShadowFunction</i>	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 subsample. 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 resolve 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 its 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

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.

For Positional Shading (POSH) there is a separate set of base addresses for Bindless and Non-Bindless Surface State called POSH Surface State Offset and POSH Bindless Surface State Base Address. They work exactly the same but allow different base addresses for positional shading.

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 associates 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.

For Positional Shading (POSH) there is a separate set of base addresses available for fetching sampler state called **POSH Bindless Surface State Base Address** and **POSH Dynamic State Base Address**.

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
<p>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.</p> <p>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.</p>	

Programming Note	
Context:	Bindless Surface State Cache Coherency
<p>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.</p> <p>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.</p>	

Programming Note	
Context:	Bindless Sampler State Cache Coherency
<p>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.</p> <p>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.</p>	

Programming Note	
Context:	State Coherency for indirect values
<p>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.</p>	

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:

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

For ASTC formats, the **ASTC Enable** bit in the RENDER_SURFACE_STATE must be set to 1. When set, the definition of the 9-bit **Surface Format** changes. The following table describes all supported formats for block-based ASTC textures.

SURFACE_FORMAT for All ASTC Formats

Value	[8] LDR/Full [7] 2D/3D [6] U8srgb /FLT16	Width 2D [5:3] 3D [5:4]	Height 2D [2:0] 3D [3:2]	Depth 2D: N/A 3D: [1:0]	Binary Form	Name	(BPE)
000h	000	0	0		000 000 000	ASTC_LDR_2D_4x4_U8sRGB	8.00
008h	000	1	0		000 001 000	ASTC_LDR_2D_5x4_U8sRGB	6.40
009h	000	1	1		000 001 001	ASTC_LDR_2D_5x5_U8sRGB	5.12
011h	000	2	1		000 010 001	ASTC_LDR_2D_6x5_U8sRGB	4.27
012h	000	2	2		000 010 010	ASTC_LDR_2D_6x6_U8sRGB	3.56

Value	[8] LDR/Full [7] 2D/3D [6] U8srgb /FLT16	Width 2D [5:3] 3D [5:4]	Height 2D [2:0] 3D [3:2]	Depth 2D: N/A 3D: [1:0]	Binary Form	Name	(BPE)
021h	000	4	1		000 100 001	ASTC_LDR_2D_8x5_ U8sRGB	3.20
022h	000	4	2		000 100 010	ASTC_LDR_2D_8x6_ U8sRGB	2.67
031h	000	6	1		000 110 001	ASTC_LDR_2D_10x5_ U8sRGB	2.56
032h	000	6	2		000 110 010	ASTC_LDR_2D_10x6_ U8sRGB	2.13
024h	000	4	4		000 100 100	ASTC_LDR_2D_8x8_ U8sRGB	2.00
034h	000	6	4		000 110 100	ASTC_LDR_2D_10x8_ U8sRGB	1.60
036h	000	6	6		000 110 110	ASTC_LDR_2D_10x10_ U8sRGB	1.28
03eh	000	7	6		000 111 110	ASTC_LDR_2D_12x10_ U8sRGB	1.07
03fh	000	7	7		000 111 111	ASTC_LDR_2D_12x12_ U8sRGB	0.89
040h	001	0	0		001 000 000	ASTC_LDR_2D_4x4_FLT16	8.00
048h	001	1	0		001 001 000	ASTC_LDR_2D_5x4_FLT16	6.40
049h	001	1	1		001 001 001	ASTC_LDR_2D_5x5_FLT16	5.12
051h	001	2	1		001 010 001	ASTC_LDR_2D_6x5_FLT16	4.27
052h	001	2	2		001 010 010	ASTC_LDR_2D_6x6_FLT16	3.56
061h	001	4	1		001 100 001	ASTC_LDR_2D_8x5_FLT16	3.20
062h	001	4	2		001 100 010	ASTC_LDR_2D_8x6_FLT16	2.67
071h	001	6	1		001 110 001	ASTC_LDR_2D_10x5_FLT16	2.56
072h	001	6	2		001 110 010	ASTC_LDR_2D_10x6_FLT16	2.13
064h	001	4	4		001 100 100	ASTC_LDR_2D_8x8_FLT16	2.00
074h	001	6	4		001 110 100	ASTC_LDR_2D_10x8_FLT16	1.60
076h	001	6	6		001 110 110	ASTC_LDR_2D_10x10_FLT16	1.28

Value	[8] LDR/Full [7] 2D/3D [6] U8srgb /FLT16	Width 2D [5:3] 3D [5:4]	Height 2D [2:0] 3D [3:2]	Depth 2D: N/A 3D: [1:0]	Binary Form	Name	(BPE)
07eh	001	7	6		001 111 110	ASTC_LDR_2D_12x10_FLT16	1.07
07fh	001	7	7		001 111 111	ASTC_LDR_2D_12x12_FLT16	0.89

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. Some formats and blending options are only correctly supported when sampler state and other state is configured correctly. See the programming notes below the following table for more details on how to support these surfaces.

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode					Border Color Mode					
				R	G	B	A		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						

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	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		R	G	1.0	1.0
R32G32_SINT			DX10/OGL	R	G	0.0	1.0					
R32G32_UINT			DX10/OGL	R	G	0.0	1.0					
R32_FLOAT_X8X24_TYPELESS	Yes		DX10/OGL	R	0.0	0.0	1.0					
X32_TYPELESS_G8X24_UINT			DX10/OGL	0.0	G	0.0	1.0					
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					
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		R	G	1.0	1.0
R16G16_SNORM			DX10/OGL	R	G	0.0	1.0		R	G	1.0	1.0
R16G16_SINT			DX10/OGL	R	G	0.0	1.0					
R16G16_UINT			DX10/OGL	R	G	0.0	1.0					
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					
R32_UINT			DX10/OGL	R	0.0	0.0	1.0					
R32_FLOAT	Yes		DX10/OGL	R	0.0	0.0	1.0		R	1.0	1.0	1.0
R24_UNORM_X8_TYPELESS	Yes		DX10/OGL	R	0.0	0.0	1.0					

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
X24_TYPELESS_G8_UINT			DX10/OGL	0.0	G	0.0	1.0					
L16A16_UNORM				L	L	L	A					
I24X8_UNORM	Yes			I	I	I	I					
L24X8_UNORM	Yes			L	L	L	1.0					
A24X8_UNORM	Yes			0.0	0.0	0.0	A					
I32_FLOAT	Yes			I	I	I	I					
L32_FLOAT	Yes			L	L	L	1.0					
A32_FLOAT	Yes			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					
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					
R16_SNORM			DX10/OGL	R	0.0	0.0	1.0					
R16_SINT			DX10/OGL	R	0.0	0.0	1.0					
R16_UINT			DX10/OGL	R	0.0	0.0	1.0					
R16_FLOAT			DX10/OGL	R	0.0	0.0	1.0		R	1.0	1.0	1.0
I16_UNORM	Yes			I	I	I	I					
L16_UNORM	Yes			L	L	L	1.0					
A16_UNORM	Yes			0.0	0.0	0.0	A					
L8A8_UNORM		Yes		L	L	L	A					
I16_FLOAT	Yes			I	I	I	I					

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode					Border Color Mode					
				R	G	B	A		R	G	B	A	
L16_FLOAT	Yes			L	L	L	1.0						
A16_FLOAT	Yes			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						
BC2_UNORM				R	G	B	A						
BC3_UNORM				R	G	B	A						
BC4_UNORM			DX10/OGL	R	0.0	0.0	1.0						
BC5_UNORM			DX10/OGL	R	G	0.0	1.0						
BC1_UNORM_SRGB				R	G	B	A						
BC2_UNORM_SRGB				R	G	B	A						
BC3_UNORM_SRGB				R	G	B	A						
YCRCB_SWAPUVV				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						
BC5_SNORM			DX10/OGL	R	G	0.0	1.0						
R16G16B16_FLOAT				R	G	B	1.0						

Surface Format Name	Shadow Map Support	Chroma Key Support	Border Color Mode	R	G	B	A	Border Color Mode	R	G	B	A
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					
EAC_RG11				R	G	0.0	1.0					
EAC_SIGNED_R11				R	0.0	0.0	1.0					
EAC_SIGNED_RG11				R	G	0.0	1.0					
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					
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					
ASTC				R	G	B	A					
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
If 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
<p>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.</p>	

Programming Note	
Context:	Quilted Surfaces
<p>Quilted Surfaces are not supported.</p>	

Programming Note	
Context:	Planar Surfaces with Disabled Sampler
<p>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.</p>	

Programming Note	
Context:	R10G10B10_FLOAT_A2_UNORM Point-Sample Precision
<p>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).</p>	

Programming Note	
Context:	Y8_UNORM
<p>Surfaces with Y8_UNORM format must programmed in RENDER_SURFACE_STATE as R8_UNORM with Shader Channel Selects set to R/R/R/1.0.</p>	

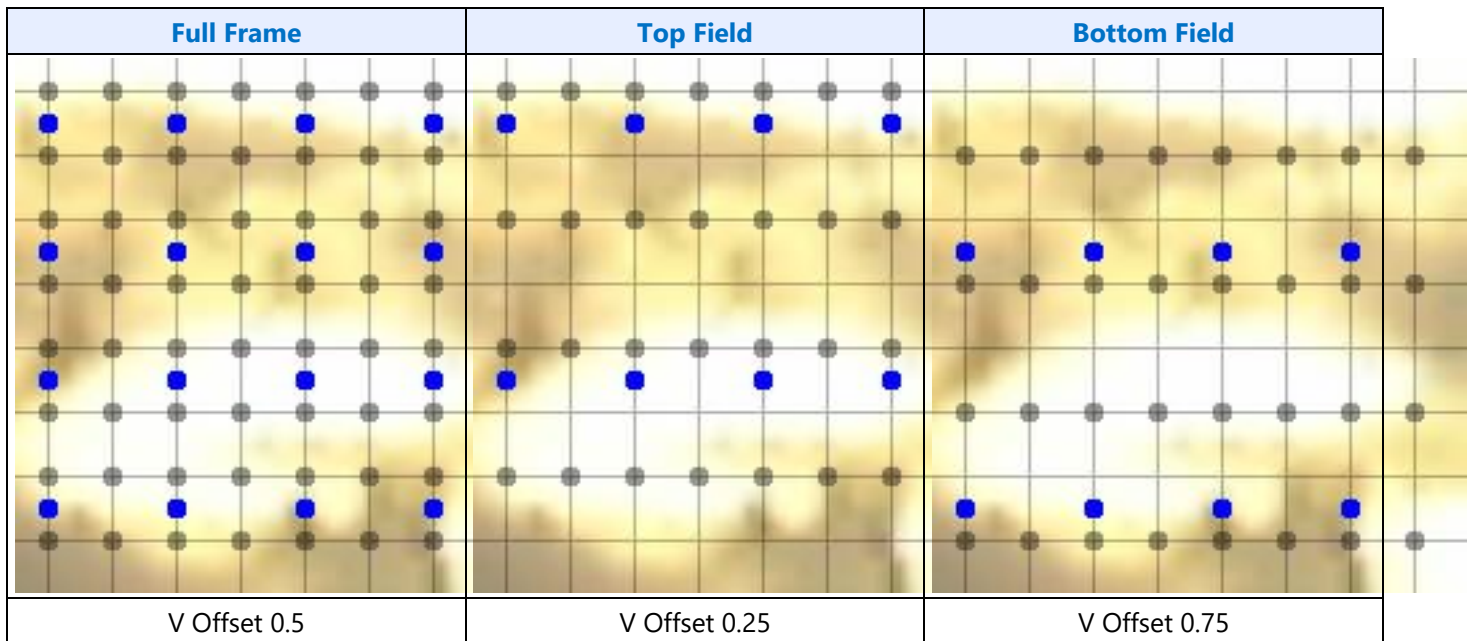
Programming Note	
Context:	Sample Tap Discard and YUV Surfaces
<p>Returned Filter Weight for Sample Tap Discard will only report the filter weight contributing texel values of the Lumina plane for YUV surface such as YCrCb.*</p>	

SURFACE_STATE for Deinterlace sample_8x8 and VME

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.



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_STATE
SAMPLER_BORDER_COLOR_STATE

If border color is used, all formats must be provided. Hardware will choose the appropriate format based on **Surface Format** and **Texture Border Color Mode**. The values represented by each format should be the same (other than being subject to range-based clamping and precision) to avoid unexpected behavior.

DWord	Bits	Description
0	31:24	Border Color Alpha Format = UNORM8
	23:16	Border Color Blue Format = UNORM8
	15:8	Border Color Green Format = UNORM8
	7:0	Border Color Red Format = UNORM8
1	31:0	Border Color Red Format = IEEE_FP
2	31:0	Border Color Green Format = IEEE_FP
3	31:0	Border Color Blue Format = IEEE_FP
4	31:0	Border Color Alpha Format = IEEE_FP
5	31:16	Border Color Green Format = FLOAT16
	15:0	Border Color Red Format = FLOAT16
6	31:16	Border Color Alpha Format = FLOAT16
	15:0	Border Color Blue Format = FLOAT16
7	31:16	Border Color Green Format = UNORM16
	15:0	Border Color Red Format = UNORM16
8	31:16	Border Color Alpha Format = UNORM16
	15:0	Border Color Blue Format = UNORM16
9	31:16	Border Color Green Format = SNORM16
	15:0	Border Color Red



DWord	Bits	Description
		Format = SNORM16
10	31:16	Border Color Alpha Format = SNORM16
	15:0	Border Color Blue Format = SNORM16
11	31:24	Border Color Alpha Format = SNORM8
	23:16	Border Color Blue Format = SNORM8
	15:8	Border Color Green Format = SNORM8
	7:0	Border Color Red Format = SNORM8

Border Color Programming for Integer Surface Formats

For integer formats, there are different possible cases depending on the bits per channel (bpc) and bits per texel (bpt) of the surface format.

Integer Surface Format - Different Types	Surface formats
32 bpc, 128 bpt case (4 types)	R32G32B32A32_UINT R32G32B32_UINT R32G32B32A32_SINT R32G32B32_SINT
16 bpc, 64 bpt case (5 types)	R16G16B16A16_UINT, R10G10B10A2_UINT X32_TYPELESS_G8X24_UINT R16G16B16_UINT R16G16B16A16_SINT R16G16B16_SINT
32 bpc, 64 bpt case (2 types)	R32G32_UINT R32G32_SINT
8 bpc, 32 bpt cases (9 types)	R8G8B8A8_UINT R8G8_UINT R8_UINT X24_TYPELESS_G8_UINT R8G8B8_UINT

Integer Surface Format - Different Types	Surface formats
	R8G8B8A8_SINT R8G8_SINT R8_SINT R8G8B8_SINT
16 bpc, 32 bpt cases (4 types)	R16G16_UINT R16_UINT R16G16_SINT R16_SINT
32 bpc, 32 bpt case (2 types)	R32_UINT R32_SINT

HW supports only 1 index for a given Sampler Border Color state and Sampler State. So, SW will have to program the table in **SAMPLER_BORDER_COLOR_STATE** at DWord offsets 16 to 19, as per the integer surface format type (depends on the bits per channel and bits per texel of the surface format). If any color channel is missing from the surface format, the corresponding border color should be programmed as zero; if the alpha channel is missing, the corresponding Alpha border color should be programmed as 1. Some of the representative cases are listed below:

Case 2: R32G32B32A32_SINT (32 bpc, 128 bpt, 4 channel, SINT)

Each of the values in the above table would have to be programmed as sint32 value.

Case 3: R32G32B32_UINT (32 bpc, 128 bpt, 3 channel)

R/G/B values would be programmed like in Case1. Alpha channel value at DWORDN+3 would have to be programmed as Integer 1.

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.	

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
<p>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.</p>	

Programming Note	
Context:	3D Sampler Messages
<p>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)</p>	

Programming Note	
Context:	Message Headers and Mid-Thread Pre-Emption
<p>A message header is required for GPGPU kernels for all sample_8x8 and sample_unorm messages to allow save/restore mechanisms to work correctly.</p>	

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
Tile Y	2D,3D,CUBE	Main_Surface_Adr[13, 12, 11, 10, 8, 7]	Main_Surface_Adr[6]
Tile Ys 8bpp	2D,3D,CUBE	Main_Surface_Adr[13, 12, 11, 10, 8, 7]	Main_Surface_Adr[6]
Tile Ys 16/32 bpp	2D,CUBE	Main_Surface_Adr[13, 12, 11, 10, 9, 8]	Main_Surface_Adr[6]
Tile Ys 32 bpp	3D	Main_Surface_Adr[13, 12, 11, 10, 8, 7]	Main_Surface_Adr[6]
Tile Ys 16	3D	Main_Surface_Adr[13, 11, 10, 9, 8, 7]	Main_Surface_Adr[6]
Tile Ys 64/128 bpp	2D,CUBE	Main_Surface_Adr[13, 12, 11, 10, 9, 8]	Main_Surface_Adr[7]
Linear Untyped	BUFFER	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. Evaluate Kernel has to have a dummy sample message before the thread EOT	

Programming Note	
Context:	Evaluate in compute context
<ol style="list-style-type: none"> 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 does 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
1) If compute shaders do evaluate, disable preemption, 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.	

Compression is done on a 128B byte cacheline. On the render path, the pairing bit determines which two 64B fragments constitute a 128B render cacheline.

Pairing bit is always bit 6

There is a potential deadlock scenario when executing texel shaders attempt to use the sampler HW while the AMFS scratch pad memory is full. To avoid this deadlock, SW must ensure that the AMFS scratch pad memory never becomes full when texel shaders issue messages to the sampler. If no texel shaders are being used, or if none of the bound texel shaders send messages to the sampler, the following workaround is not required.

To ensure that FIFO memory is never filled, SW must perform a combination of:

1. Increasing the FIFO memory size
2. Limiting the number of PS threads allowed to run simultaneously. This can be done by setting the Maximum Number of Fused Threads Per PSD within the **3DSTATE_PS** instruction.

In general, the worst-case amount of FIFO memory used can be calculated with the following:

$$\text{FIFOMemUsed} = \text{MaxFusedThreadsPerPSD} * \text{NumEvalMsgsPerPS} * \text{AMFSExpansionFactor} * \text{SIMDLanesPerEU} * \text{NumEUsFused}(2) * \text{NumPSDPerGT}(6) * \text{BytesPerAMFSMsg}(64)$$

MaxFusedThreadsPerPSD: Can be controlled by SW, maximum value of 56 physical threads.

NumEvalMsgsPerPS: Can sometimes be determined by SW (during shader compile). If all of the eval messages are outside of flow control this is an easy calculation. When evaluate messages are inside of branches or loops this may be impossible to determine. In this case the API / extension should have an upper limit of the number of evaluate messages allowed to execute within a single PS invocation.



AMFSExpansionFactor: Refer to the following table which shows the worst-case number of AMFS transactions which can be generated from a single Evaluate message in 3D Sampler:

Sample SIMD16 Operation Type	Maximum Number of AMFS Transactions (Worst-Case)	Comment
Typical Point-Sample/Bilinear	256	This assumes all accesses Edge of Map
2D MIP Linear	512	This assumes all accesses on Edge of Map
Cube MIP Linear on Corner	1024	This assumes all accesses are to Corner of Cube. Enabling Low-Quality Cube Corner Mode will not impact AMFS quality but will limit number of AMFS transactions to 1024. Enabling Normal-Quality Cube Corner mode will increase this number to 4096.
X16 Anisotropic Filtering (w/ MIP Linear)	4096	This assumes all access are edge of the map. Limiting anisotropy will reduce this number commensurately.

SW cannot always determine the filtering modes being used, but there may be API limits or extension hints that will help inform the driver of the parameters being used.

SIMDLanesPerEU: SIMD width of kernel executing (8 / 16).

NumEUsFused: 2 (every 2 EUs are considered a single "fused-EU")

NumPSDPerGT: Number of Pixel Shader Dispatchers (1 per dual-sub-slice) per GPU. For GT2 this is 6.

BytesPerAMFSMsg: 64 Bytes are used per AMFS message.

Note: The above calculation is a worst-case involving very specific texture coordinates being used for the evaluate messages. In most realistic usage scenarios the actual memory used will be considerably less.

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.). The lower 11 bits of the Extended Message Descriptor are defined by the Extended Message Descriptor for the Execution Unit.

The message descriptor is sent in parallel with the message payload (and header if used).

Descriptor
Message Descriptor - Sampling Engine
Extended Message Descriptor - Sampling Engine
Extended Message Descriptor - Execution Unit

Programming Note	
Context:	3D Sampler Messages
<h3>Restrictions for Message Descriptors</h3> <p>Use of any message to the Sampling Engine function with the End of Thread bit set in the message descriptor is not allowed.</p>	

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:

Contents
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
sample_unorm1
sample_8x81

¹ For SIMD32/SIMD64 Media-type messages such as sample_unorm, sample_8X8 and deinterlace see the Media Sampler SIMD32/64 section for a description of the payloads of these messages and the expected behavior.

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 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.	

Sample Message Types

Sample Message Definition

Message Type	Description
sample_*	<p>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.</p> <p>The sample message enables filtering/blending operations (e.g. MAP_LINEAR) and addressing modes (e.g. Mirror) which are controlled by SAMPLER_STATE.</p>

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 message 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	<p>The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The <i>r</i> 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 <i>u</i>, <i>v</i>, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The <i>r</i> coordinate contains the faceid, and the <i>r</i> gradients are ignored by hardware.</p> <p>Previously known as sample_g.</p>
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.

Programming Note

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.

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 *mlo*d 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, SURFTYPE_3D, or SURFTYPE_CUBE.

Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.

sample_d: The *mlo*d 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
<p>sample_killpix:</p> <p>The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE.</p> <p>sample_killpix is supported only in SIMD8 mode.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p>
<p>sample_killpix:</p> <p>If the sampler is disabled via the Sampler Disable bit in the SAMPLER_STATE, the Kill Pixel Mask returned will be undefined.</p>

Restrictions and Programming Notes for sample_min, sample_max

Programming Note
<p>The Surface Type of the associated surface must be SURFTYPE_2D and Surface Array must be disabled.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1</p>
<p>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:</p> <ul style="list-style-type: none"> • Sampler state address mode for q direction specifies D3D11_TEXADDRESS_BORDER • Address mode for U and V do not specify D3D11_TEXADDRESS_BORDER
<p>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.</p>

Restrictions and Programming Notes for sample_lz

Programming Note
<p>sample_lz:</p> <p>The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D</p>

Id Message Types

Id Message Definition

Message Type	Description
ld_*	<p>The surface is sampled using a default sampler state.</p> <p>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.</p> <p>If the message doesn't include an LOD parameter, the message samples from LOD 0.</p> <p>For ld_* message types, the sampler state is defaulted as follows:</p> <ul style="list-style-type: none"> • 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 without an alpha channel, which return a value of one in the alpha channel. <p>The ld* family of messages do not perform any blending operation.</p> <p>Additional parameters are needed for the variants of ld defined below.</p> <p>Potentially lower power than using sample_l with Filter Mode=MAP_NEAREST because coordinates do not need to be converted from floating-point.</p>

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	Basic ld operation but to a multi-sampled surface. Includes additional parameters for MCS values (see ld_mcs below) and Si (sample index) for each pixel in the message. Can be used for all multi-sampled surfaces up to X8 MSAA amd for X16 MSAA in cases where the upper half of the sub-samples (planes 8 thru 15) all reference plane0 (see ld_mcs to see how determining X16 MSAA viability is determined).
ld2dms_w	<p>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.</p> <p>For SIMD8/SIMD16 the mcsh parameters is only used for X16 MSAA, and should be all 0's for X8 MSAA or lower depth.</p>
ld2dss	Same as ld2dms, but does not require MCS, and uses only ssi parameter to pick same sub-slice for all pixels. The ssi parameter defines the sample slice that will be sampled from. Refer to the multisample

Message	Description
	storage format in the GPU Overview volume for more details.
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/ld2dms_w/ld2dss message to sample from the multi-sampled surface.

Restrictions and Programming Notes for ld

Programming Note
<p>ld:</p> <p>The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1</p>

Restrictions and Programming Notes for ld_lz

Programming Note
<p>ld_lz:</p> <p>The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the ld message.</p>

Restrictions and Programming Notes for ld_mcs

Programming Note
<p>ld_mcs:</p> <p>The Surface Type of the surface associated with the auxiliary surface must be SURFTYPE_2D.</p>
<p>ld_mcs:</p> <p>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.</p>
<p>ld_mcs:</p> <p>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).</p>
<p>ld_mcs:</p> <p>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.</p>

Programming Note

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 ld2dms

Programming Note

ld2dms:

The sampled surface type must not be MULTISAMPLECOUNT_1

The sample surface type must not be MULTISAMPLECOUNT_16 any of the upper 8 planes contain valid data.

The Surface Type of the associated surface must be SURFTYPE_2D.

ld_2dms:

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).

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:

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_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 Id_2dss_w

Programming Note

Id_2dss:

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).

Restrictions and Programming Notes for Id

Programming Note

Id:

The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the Id message.

Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1

Restrictions and Programming Notes for Id_lz

Programming Note

Id_lz:

The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_BUFFER for the Id message.

Restrictions and Programming Notes for Id_mcs

Programming Note

Id_mcs:

The Surface Type of the surface associated with the auxiliary surface must be SURFTYPE_2D.

Id_mcs:

The Id_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, Id_mcs behaves like the Id message. If Id_mcs is issued on a surface with Auxiliary Surface Mode not set to AUX_MCS, this message returns zeros in all channels.

Id_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).

Id_mcs:

If Id_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 Id2dms_w message to

Programming Note

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 ld2dms

Programming Note

ld2dms:

The sampled surface type must not be MULTISAMPLECOUNT_1

The sample surface type must not be MULTISAMPLECOUNT_16 any of the upper 8 planes contain valid data.

The Surface Type of the associated surface must be SURFTYPE_2D.

ld_2dms:

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).

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:

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_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_2dss_w

Programming Note

ld_2dss:

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).

gather4 Message Types

Definitions

Message Type	Description				
gather4_*	<p>The surface is sampled using MAP_LINEAR filtering, regardless of the filtering mode specified in the sampler state and with LOD forced to zero of the visible resource.</p> <p>However, the samples are not filtered. Instead, the selected color channel the four contributing texels are returned directly in the sample's corresponding four channels as follows:</p> <table border="1" style="margin-left: 40px;"> <tr> <td>upper left sample = alpha channel</td> <td>upper right sample = blue channel</td> </tr> <tr> <td>lower left sample = red channel</td> <td>lower right sample = green channel</td> </tr> </table> <p>Returned color channel is selected through a message field: Gather4 Source Channel Select</p> <p>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.</p>	upper left sample = alpha channel	upper right sample = blue channel	lower left sample = red channel	lower right sample = green channel
upper left sample = alpha channel	upper right sample = blue channel				
lower left sample = red channel	lower right sample = green channel				
gather4_*	<p>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.</p>				

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.
gather4_po	Similar to gather4, but includes offu and offv parameters, which contain texel-space offsets that override the U/V Offset fields in the message header. Unlike the message header fields however, these offsets have a wider range [-32,+31], and can differ per pixel or sample. The format of the data is 32-bit 2's complement signed integer, but hardware only interprets the least significant 6 bits of each value, treating it as a 6-bit 2's complement signed integer
gather4_po_c	This is a combination of gather4_c and gather4_po, and will return data only on the Red Channel (Gather4 Source Channel Select must be set to Red), but uses offu and offv to shift the position on the surface.

Restrictions and Programming Notes for gather4:

Programming Note
<p>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. Using DX9 Border Color Mode when either of the following is true will yield undefined results:</p> <ul style="list-style-type: none"> 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.
<p>gather4: If Surface Format is R10G10B10_SNORM_A2_UNORM and Gather4 Source Channel Select is alpha channel, the returned value may be incorrect.</p>
<p>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.</p>

Restrictions and Programming Notes for gather4_c:

Programming Note
<p>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. Using DX9 Border Color Mode and either of the following is undefined:</p> <ul style="list-style-type: none"> 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.
<p>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.</p>

Restrictions and Programming Notes for gather4_po:

Programming Note
<p>gather4_po: The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p> <p>Using DX9 Border Color Mode and either of the following is undefined:</p> <ul style="list-style-type: none"> Any applicable Address Control Mode (depending on Surface Type) is set to TEXCOORDMODE_CLAMP_BORDER or TEXCOORDMODE_HALF_BORDER. <p>Surface Type is SURFTYPE_CUBE and any Cube Face Enable is disabled.</p>
<p>gather4_po: 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.</p>

Restrictions and Programming Notes for gather4_po_c:

Programming Note
<p>gather4_po_c: The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>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</p> <p>The channel selected by the Gather4 Source Channel Select field in the message header must be set to Red.</p> <p>Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.</p> <p>Using DX9 Border Color Mode and either of the following is undefined:</p> <ul style="list-style-type: none"> 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.
<p>gather4_po_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.</p>

sampleinfo Message Type

sampleinfo Message Definition

Message Type	Description
sampleinfo	<p>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 Index are ignored.</p> <p>The Surface Type of the associated surface must be SURFTYPE_2D or SURFTYPE_NULL.</p>



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: The return format for LOD message must be 32-bits.

resinfo Message Type

resinfo Message Definition

Message Type	Description
resinfo	<p>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.</p> <p>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.</p> <p>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).</p> <p>The Sampler State Pointer and Sampler Index are always ignored.</p> <p>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.</p>
resinfo	<p>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.</p> <p>The Sampler State Pointer and Sampler Index are always ignored.</p>

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



Surface Type	Red	Green	Blue	Alpha
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	<p>The texture caches in the sampling engine are invalidated. This includes all levels of texture cache, however the state caches are not invalidated.</p> <p>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.</p>

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. This message must be sent as a SIMD32/64 type message with header only (no additional message contents)
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.

Media Message Types

The 3D sampler supports specific message types which are intended for use by the Media sampler. These messages include:

Message
sample_unorm
sample_8X8

Media message types are described in more detail, with payload information defined in the Media Sampler Section.

sample_unorm Message Types

sample_unorm Message Definition

Message Type	Description
sample_unorm_*	<p>The surface is sampled using the indicated sampler state. 32 contiguous pixels in a 8-wide by 4-high block are sampled. The U and V addresses for the upper left pixel are delivered in this message along with Delta U and Delta V floating-point parameters and . Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are computed as follows:</p> $U(x,y) = U(0,0) + \text{DeltaU} * x$ $V(x,y) = V(0,0) + \text{DeltaV} * y$ <p>Typically this message and its variants are used for media applications.</p>

Supported Variants:

Message	Description
sample_unorm	Basic sample_unorm behavior as described above.
sample_unorm_RG	Similar to sample_unorm, but only the Red and Green Channels are returned because there is an implied write mask.
sample_unorm_killpix	This message is identical to the sample_unorm message except it returns a kill pixel mask in addition to the selected channels in the writeback message. 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. All restrictions of the sample_unorm message apply to this message also.
sample_unorm_RG_killpix	This message is identical to the sample_unorm_RG message except it returns a kill pixel mask in addition to the selected channels in the writeback message. 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. All restrictions of the sample_unorm message apply to this message also.

Restrictions and Programming Notes for sample_unorm, sample_unorm_RG, sample_unorm_killpix, sample_unorm_RG_killpix:

Programming Note
<p>The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>The Surface Format of the associated surface must be UNORM with ≤ 8 bits per channel.</p> <p>The MIP Count, Depth, Surface Min LOD, Resource Min LOD, and Min Array Element of the associated surface must be 0.</p> <p>The Min and Mag Mode Filter must be MAPFILTER_NEAREST or MAPFILTER_LINEAR.</p> <p>The Mip Mode Filter must be MIPFILTER_NONE.</p> <p>The TCX and TCY Address Control Mode cannot be any of:</p> <ul style="list-style-type: none"> • TEXCOORDMODE_CLAMP_BORDER • TEXCOORDMODE_HALF_BORDER • TEXCOORDMODE_MIRROR_ONCE • TEXCOORDMODE_WRAP <p>Map Width must be ≥ 4.</p> <p>Map Height must be ≥ 4</p> <ul style="list-style-type: none"> • Number of Multisamples on the associated surface must be MULTISAMPLECOUNT_1.
<p>Supports additional Surface formats: PLANAR_420_16, R16_UNORM, R16G16_UNORM, R16G16B16A16_UNORM</p>
<p>Map Width must be ≥ 16</p> <p>Map Height must be ≥ 16</p>

sample_8x8 Message Type

sample_8x8 Message Definition

Message Type	Description
sample_8x8	<p>The surface is sampled using an optional 8x8 filter, using state defined in SAMPLER_STATE and SAMPLER_8x8_STATE.</p> <p>The input consists of 64 contiguous pixels in a 16-wide by 4-high arrangement. The address control mode behaves as clamp mode. The U and V addresses for the upper left pixel are delivered in this message along with a Delta U and Delta V parameter. Given a pixel at (x,y) relative to the upper left pixel (where (0,0) is the upper left pixel), the U and V for that pixel are computed as follows:</p> $U(x,y) = U(0,0) + \text{DeltaU} * x + U_{2^{\text{nd}}_Derivative} * x * (x - 1)/2$ $V(x,y) = V(0,0) + \text{DeltaV} * y + V_{2^{\text{nd}}_Derivative} * y * (y - 1)/2$

Supported Variants:

None.

Restrictions and Programming Notes for sample_8x8:

Programming Note
<p>The Surface Type of the associated surface must be SURFTYPE_2D.</p> <p>The Surface Format of the associated surface must be UNORM with ≤ 10 bits per channel.</p> <p>$\Delta V * Height$ of the associated surface must be less than 16.0.</p> <ul style="list-style-type: none"> <i>Map Width</i> must be ≥ 4.
<p>Supports functionality such as convolve, MinMax, MinMaxFilter, Dilate, Erode, BoolCentroid, and Centroid have been added on top of the existing sample_8x8. The convolve, MinMax, and MinMaxFilter are 16-wide X 4-high; the rest have variable size depending on the message. See Media Sampler section for a description of how these modes work.</p>

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
SIMD32/64	Media Operations on 32 or 64 pixels

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.	

Programming Note	
Context:	3D Sampler Messages - Message Format
The Cache_Flush message is SIMD32, but has no payload, only a header. So, the length is 1 for Cache_Flush	

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. The value of k shown in the response length is 1 if killpix is enabled and 0 otherwise.

SIMD Mode	Return Precision (bits per channel)	Response Length
SIMD8	32	4+k
SIMD8	16	2**
SIMD16	32	8*
SIMD16	16	4*

Notes for Determining Response Lengths Table

Symbol	Note
*	For SIMD16, phases in the response length are reduced by 2 for each channel that is masked. SIMD16 messages with six or more parameters exceed the maximum message length allowed, in which case they are not supported. This includes some forms of sample_b_c, sample_l_c, and gather4_po_c message types. Note that even for these messages, if 5 or fewer parameters are included in the message, the SIMD16 form of the message is allowed. SIMD16 forms of sample_d and sample_d_c are not allowed, regardless of the number of parameters sent.
**	For SIMD8*, phases in the response length are reduced by 1 for each channel that is masked.
***	All SIMD16H messages have the same limitations as the corresponding SIMD16 message would, even though maximum message length is not exceeded. The SIMD16H version of ld2dms_w is an exception. It allows up to 7 parameters.

SIMD32/SIMD64 Special Cases for Response Length

The SIMD32/SIMD64 modes have special response lengths depending on the message type.

Mnemonic	Payload Layout	Response Length
sample_unorm	Pixel Shader	8**
sample_unorm + killpix	Pixel Shader	9**
deinterlace	Pixel Shader	+
sample_unorm	Media	8**
sample_unorm + killpix	Media	9**
sample_8X8	Media	16*
cache_flush	no payload	1

** For sample_unorm, phases in the response length are reduced by 2 for each channel that is masked.

* For sample_8x8, phases in the response length are reduced by 4 for each channel that is masked.

+ For deinterlace, response length depending on certain state fields. Refer to writeback message definition for details.

If **Header Present** is *disabled*, **Response Length** can be set to values in the table below for SIMD8 and SIMD16 messages other than *sample+killpix*. The setting of **Response Length** determines 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*	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*	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



SIMD Mode	Return Format	Response Length	Alpha Write Channel Mask	Blue Write Channel Mask	Green Write Channel Mask	Red Write Channel Mask
SIMD8*	32-bit/ 16-bit	1	1	1	1	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 Id*, 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	8	9	10	
00000	sample	u	v	r	ai	mlod							
00001	sample_b	bias	u	v	r	ai	mlod						
00010	sample_l	lod	u	v	r	ai							
00011	sample_c	ref	u	v	r	ai	mlod						
00100	sample_d	u	dudx	dudy	v	dvdx	dvdy	r	drdx	drdy	ai	mlod	
00101	sample_b_c	ref	bias	u	v	r	ai						
00110	sample_l_c	ref	lod	u	v	r	ai						
00111	ld	u	v	lod	r								
01000	gather4	u	v	r	ai								

Encoding	Mnemonic	Parameters											Comment	
01001	LOD	u	v	r	ai									
01010	resinfo	lod												
01011	sampleinfo													
01100	sample+killpix	u	v	r										
10000	gather4_c	ref	u	v	r	ai								
10001	gather4_po	u	v	offu	offv	r								
10010	gather4_po_c	ref	u	v	offu	offv	r							
10100	sample_d_c	ref	u	dudx	dudy	v	dvdx	dvdv	r	drdx	drdy	ai		
11000	sample_lz	u	v	r	ai									
11001	sample_c_lz	ref	u	v	r	ai								
11010	ld_lz	u	v	r										
11100	ld2dms_w	si	mcs1	mcs0	u	v	r	lod						SIMD8and SIMD16 Only
11100	ld2dms_w	si	mcs0	mcs1	mcs2	mcs3	u	v	r	lod				SIM8H and SIMD16H Only
11101	ld_mcs	u	v	r	lod									SIMD8 and SIMD16 Only
11101	ld_mcs	u	v	r	lod									SIMD8H and SIMD16H Only
11110	ld2dms	si	mcs	u	v	r	lod							
11111	ld2dss	ssi	u	v	r	lod								

The table below shows the supported messages and their formats for SIMD32/SIMD64 Messages:

Encoding	Mnemonic	Comment
00000	sample_unorm	
00001	sample_unorm+killpix	
01000	deinterlace	
01010	sample_unorm	
01011	sample_unorm + killpix	
01100	sample_8x8	
11111	cache_flush	

Programming Note

Context: 3D Sampler Messages - Message Format

The ld, ld_mcs, ld2dss, resinfo, gather4_po utilize integer operands and must not be used with SIMD8H SIMD16H which are for 16-bit floating-point operands.

Programming Note	
Context:	3D Sampler Messages - Message Format
For output control modes other than "16 bit Full", the response lengths are reduced accordingly. Refer to the writeback message format definitions to determine the correct response length for each case.	

Programming Note	
Context:	Min LOD Parameter
To ensure optimal performance, the Min LOD (m lod) parameter should not be included in a <i>sample</i> or <i>sample_b</i> message unless one or more of the subspans requires a non-zero Min LOD.	

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 Id 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

DWord	Bits	Description
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 Id 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 Id and resinfo message types.
	15:0	Subspan 3, Pixel 2 (lower left) Parameter 0
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 Id 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	Subspan 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	Subspan 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	Subspan 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

DWord	Bit	Description
W1.1	31:0	Subspan 2, Pixel 1 (upper right) Red
W1.0	31:0	Subspan 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	Subspan 3, Pixel 0 (upper left) Red

DWord	Bit	Description
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	Subspan 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	Subspan 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	Subspan 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	Subspan 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	Subspan 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.
W4.0	31:16	Dispatch Pixel Mask: For this 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**.

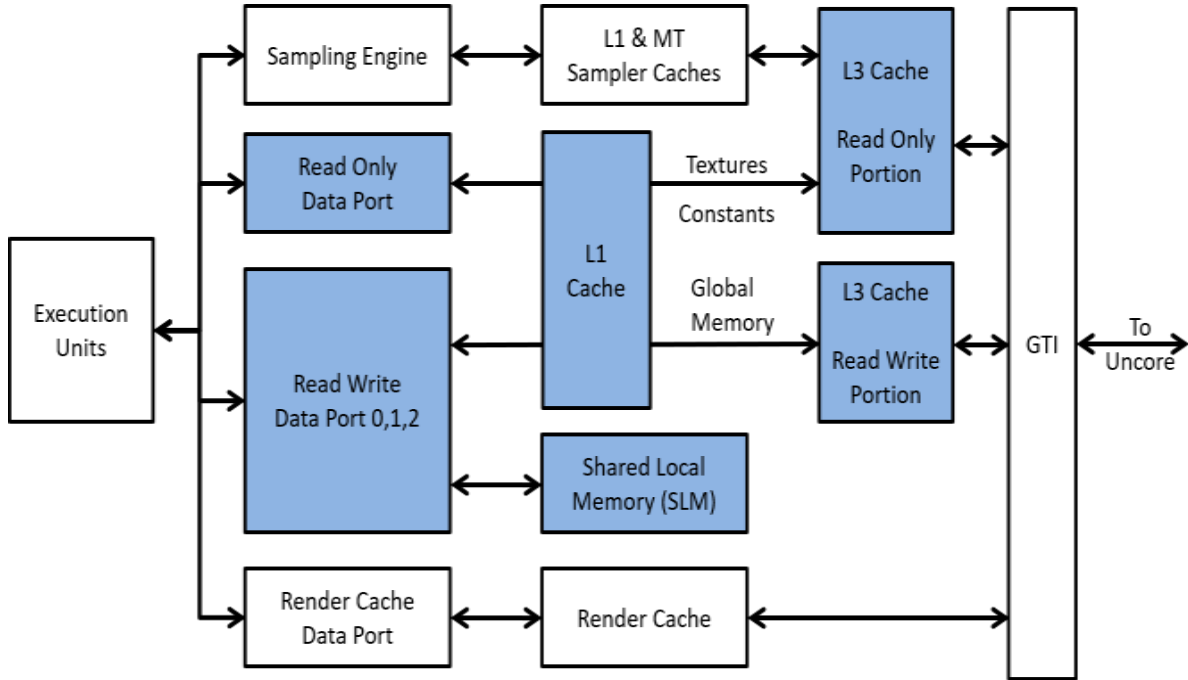
DWord	Bits	Description
W0.7:4		Reserved
W0.3	31:16	Subspan 1, Pixel 3 (lower right) Red: Specifies the value of the pixel's red channel. Format = IEEE Half Float, S15 signed 2's comp integer, or U16 unsigned integer.
	15:0	Subspan 1, Pixel 2 (lower left) Red
W0.2	31:16	Subspan 1, Pixel 1 (upper right) Red
	15:0	Subspan 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	Subspan 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 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.

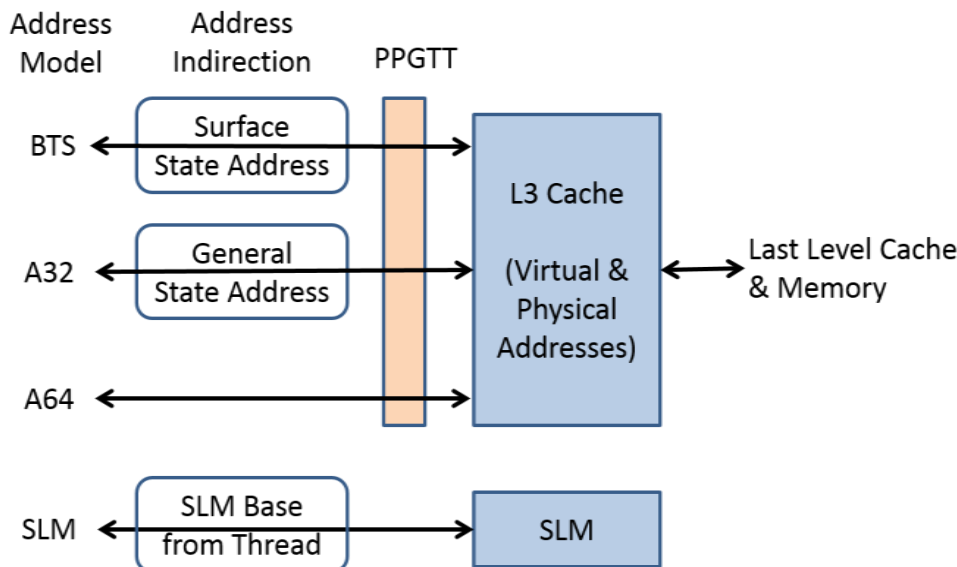
When memory access ordering is required and not guaranteed, either in the same thread or with different threads, software must ensure the ordering using fence operations or atomic operations. The Memory Fence message can be used to block a thread until all previous messages issued to that data port cache 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

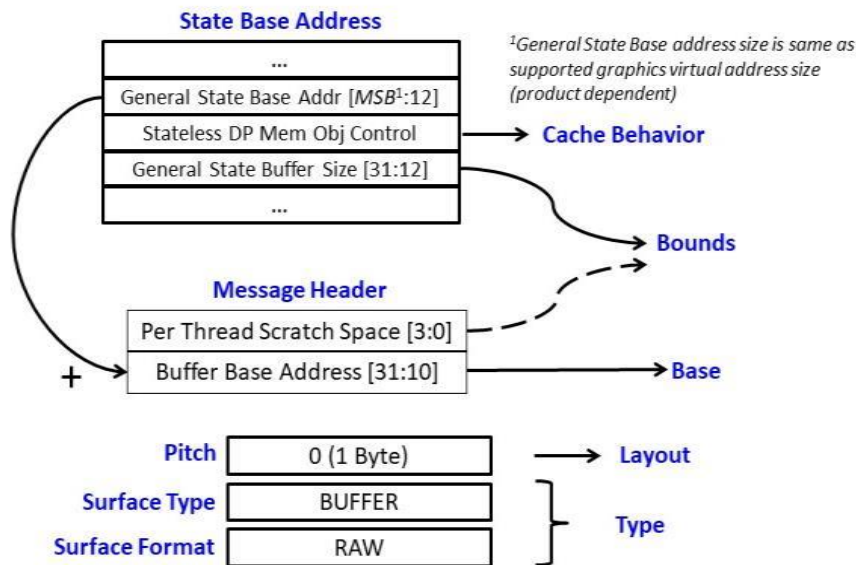
In the A32 model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = RAW
- Vertical Line Stride = 0
- Surface Base Address = General State Base Address + (optional) Buffer 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

A32 Stateless Memory Characteristics



Stateless Model Aliases

For products where IA coherency is supported, the stateless aliases 255 and 253 provide a means of SW controlling the coherency properties of an access. The coherency property is ensured for that access only. Typically, SW uses the same coherency type for all access to the same address. Proper fencing is required to ensure that reads and writes are visible. L3UC forces the addressed cache lines out of L3 and the cycles are directly conducted to LLC. This provides a capability for ensuring coherency on a particular location without having to fence all the other cycles.

Binding Table Index	Type	Description
255	IA Coherent	Coherent within GFX and coherent within the entire IA cache memory hierarchy.
253	Non-Coherent	Coherent within GFX for the same cache type (L3, Sampler, or Render).

Programming Note	
Context:	A32 Stateless Model
<ul style="list-style-type: none"> The constant, sampler, and render caches are always non-coherent. A32 messages that are stateless and specified without a BTI (Hword Read/Write and all messages on SFID_DP_DC2) are IA coherent. 	

Programming Note

If STATE_COMPUTE_MODE instruction sets Force Non-Coherent Mode, then all A32 accesses by Data Port messages are non-coherent.

A64 Flat/Stateless Model

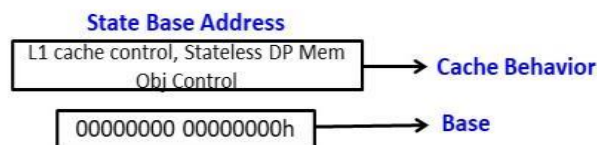
In the A64 model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- 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

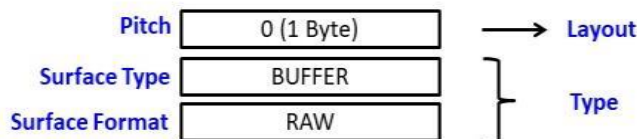
This model is primarily intended for programmable shader programs.

A64 Stateless Memory Characteristics

A64 Stateless Memory Characteristics



Bounds [Canonical address check] After address calculation, bits [63:VA_MSB+1] must match bit [VA_MSB].
 VA_MSB is the MSB of the supported graphics virtual address size.



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	
Context:	A64 Stateless Model
A64 messages that are stateless and specified without a BTI (Hword Read/Write and all messages on SFID_DP_DC2) are IA coherent.	

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

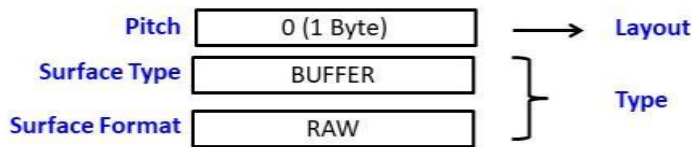
Due to the predefined surface state attributes for the SLM, only a subset of the data port messages can be used.

SLM Memory Characteristics

SLM Memory Characteristics



TG SLM Size is allocated with new thread groups with values from 1KB to *Max_TG_SLM_size*, in discrete steps. The steps and *Max_TG_SLM_size* are product dependent.



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
32KB
64KB

Specifying SLM Access

Specifying SLM Access
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.

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

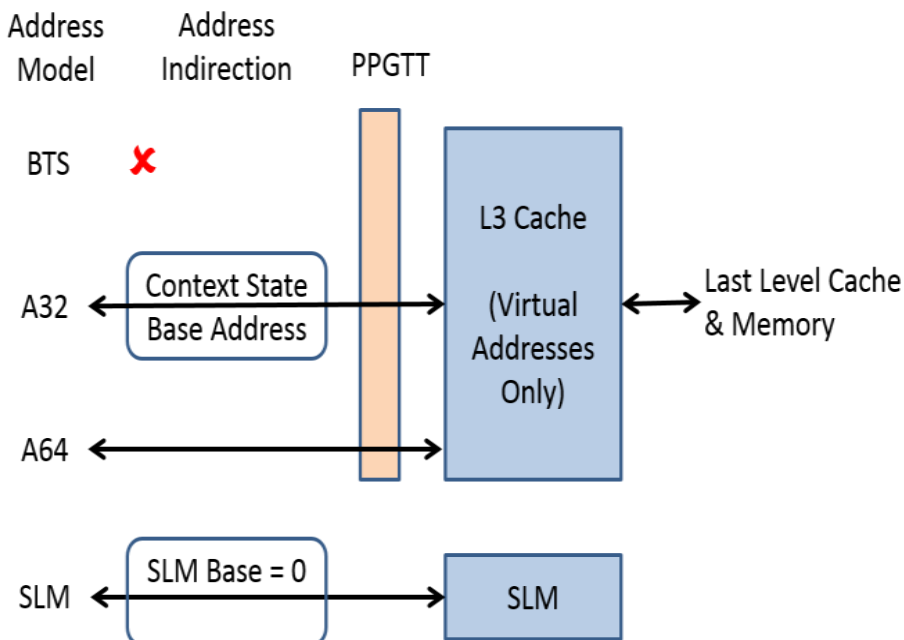
Unified Return Buffer memory (URB) is a high bandwidth memory that is not backed up by system memory. The URB contents are shared between all active threads in the same context. Its contents are uninitialized after creation, and its contents disappear when deallocated.

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
<p>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.</p> <p>However, Hword Scratch Read/Write messages do not override the A32 base address, and should not be used during SIP.</p>

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.
Address Alignment	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.
Address Calculation	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.
Bounds Checking	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).
Canonical Address	The calculated virtual memory address, formed from each calculated offset and the Address model's Base Address, could cross the canonical address boundary.

Characteristic	Overview						
<p>Checking</p>	<p>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.</p> <p>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.</p>						
<p>Tiled Resources Checking</p>	<p>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).</p> <p>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:</p> <ul style="list-style-type: none"> • Null tiles, which are treated as Out-of-Bounds • Invalid tiles, which are treated as page faults <p>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.</p> <table border="1" data-bbox="362 930 1494 1129"> <thead> <tr> <th colspan="2" data-bbox="362 930 1494 972">Programming Note</th> </tr> </thead> <tbody> <tr> <td data-bbox="362 972 683 1020">Context:</td> <td data-bbox="683 972 1494 1020">Cached Read of a Null Tile</td> </tr> <tr> <td colspan="2" data-bbox="362 1020 1494 1129">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> </tbody> </table>	Programming Note		Context:	Cached Read of a Null Tile	If a read from a Null tile gets a cache-hit in a virtually-addressed GPU cache, then the read may not return zeroes.	
Programming Note							
Context:	Cached Read of a Null Tile						
If a read from a Null tile gets a cache-hit in a virtually-addressed GPU cache, then the read may not return zeroes.							
<p>Fault Handling</p>	<p>Virtual memory addresses for data ports are translated through the PPGTT. If a fault occurs in 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.</p> <p>SLM accesses do not fault.</p> <p>See Read/Write Ordering in this volume for a description of how memory ordering is impacted by a page fault.</p>						
<p>Out-of-Bounds handling</p>	<p>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.</p>						
<p>Cache Behavior</p>	<p>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.</p> <p>Each Address Model specifies the coherency model for the read/write data cache ports.</p> <ul style="list-style-type: none"> • 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 Stateless Model Aliases) 						

Characteristic	Overview
	<p>in A32 Stateless Model in this volume for details)</p> <ul style="list-style-type: none"> • SLM accesses are localized and not cached <p>The behavior of writing cache data to memory (e.g. write back or write through modes) is specified by the MEMORY_OBJECT_CONTROL_STATE.</p> <ul style="list-style-type: none"> • 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 <p>When PPGTT is enabled, the page table entry overrides the LLC/eDRAM cache controls supplied by the MEMORY_OBJECT_CONTROL_STATE.</p>

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, 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.

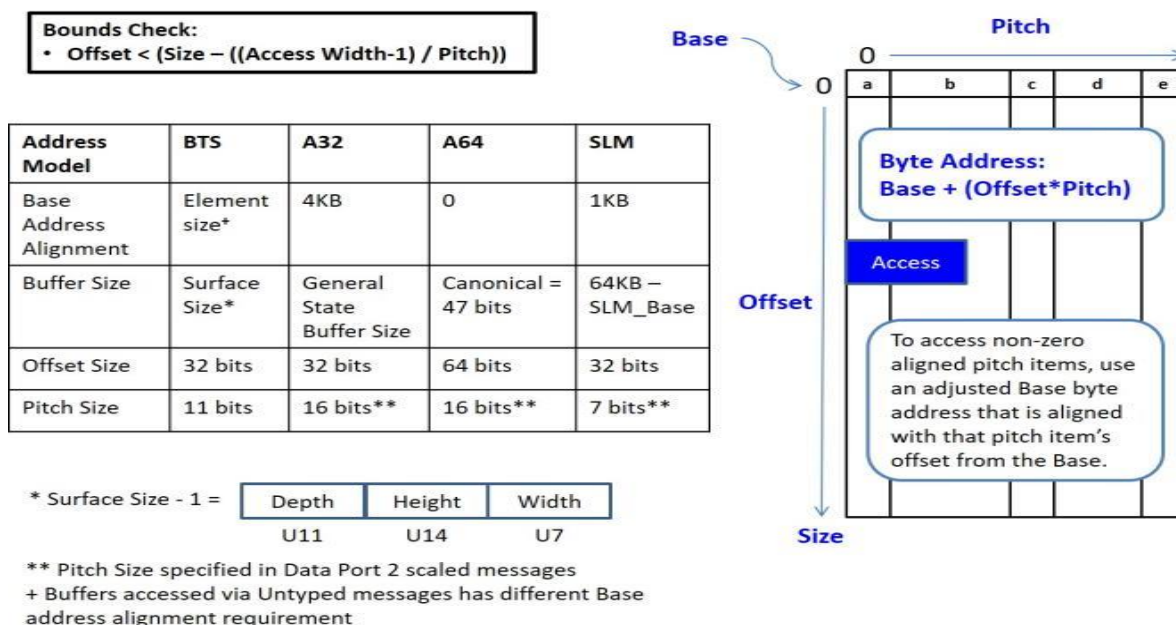
Programming Note	
Context:	Surfaces Types
For data port messages, compressed surface formats for MSAA and Media surfaces must be Y Major tiled (not Linear, X Major, or W major).	

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

BUFFER: Array of Structures



Untyped Message U Offset Alignment

For untyped messages accessing SURFTYPE_BUFFER surface types, the U address (byte offset) must be aligned to the data access width (for example, DWords must have the low 2 bits as zeros).

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.

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

- Bounds Check:**

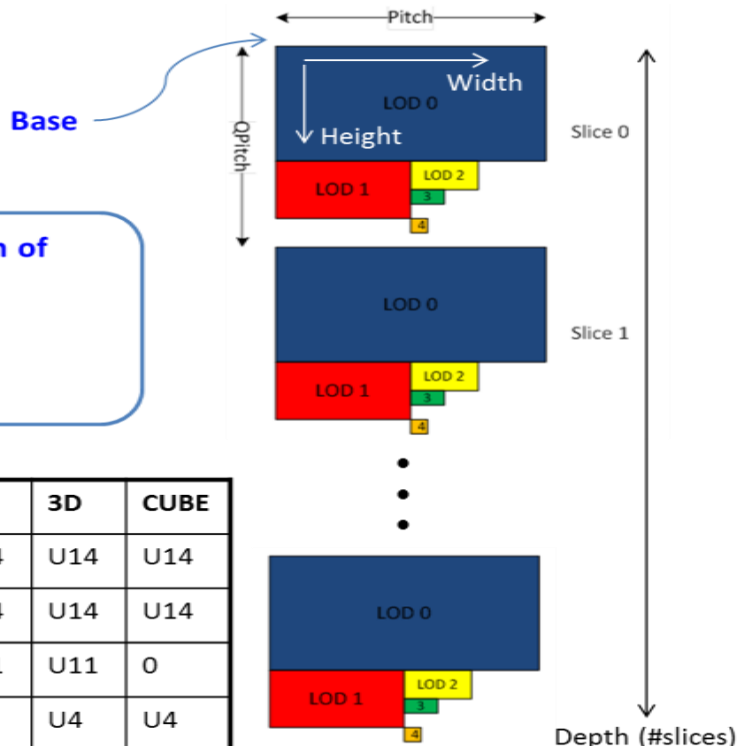
 - U < Width
 - V < Height
 - R < Depth
 - LOD < MIPCount

Example of Address Calculation of Surface[U,V,R,LOD]:

$$\text{Base} + (R * QPitch) + (V * Pitch) + U + \text{LOD_Offset}$$

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
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

In reference to WA

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
<ul style="list-style-type: none"> 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	<i>Not supported</i>
RAW16	Any	16 bit copy, zero extend MSBs	<i>Not supported</i>
RAW8	Any	8 bit copy, zero extend MSBs	<i>Not supported</i>

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

Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)	Data Returned By Typed Read			
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
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
0C8	R8G8B8A8_UNORM_SRGB	32
0C9	R8G8B8A8_SNORM	32
0CA	R8G8B8A8_SINT	32
0CB	R8G8B8A8_UINT	32
0CC	R16G16_UNORM	32
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

Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
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

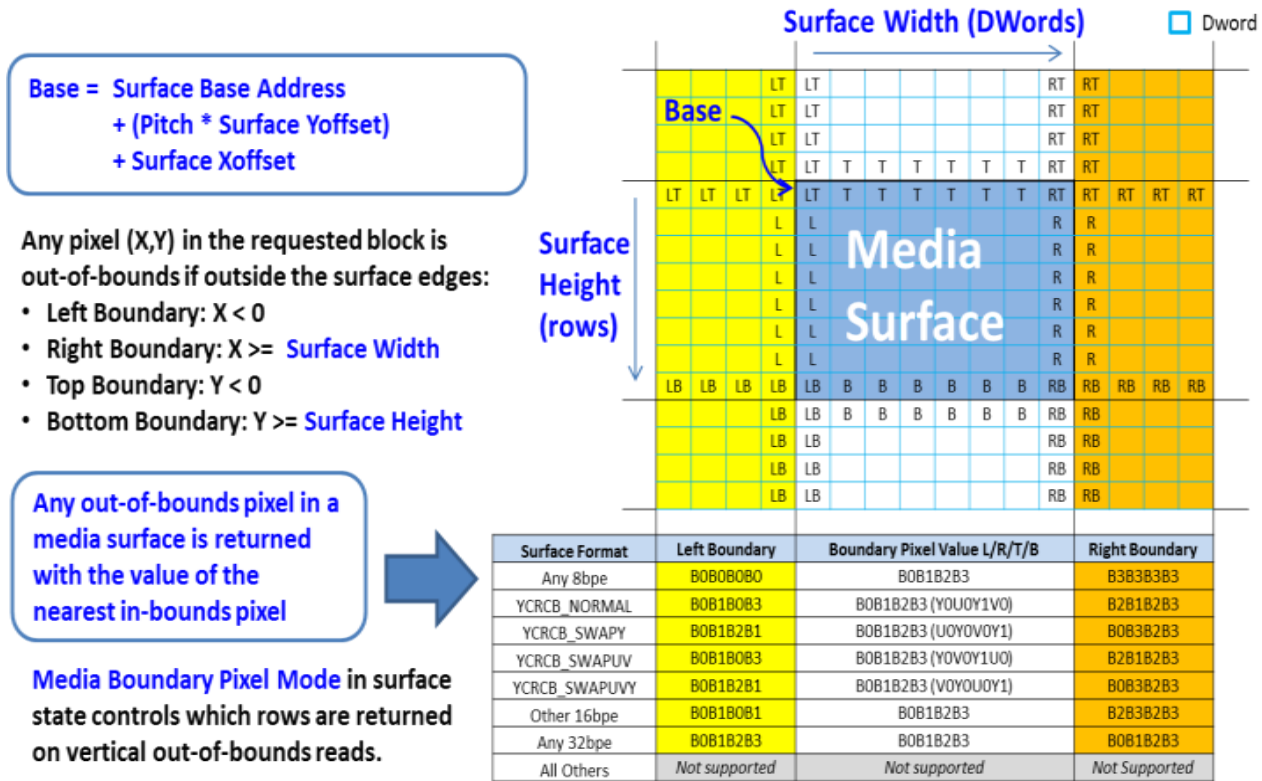
Programming Note	
Context:	Surface Formats
Indirectly supported surface formats are treated as R8_UINT for the 8 BPE formats, R16_UINT for 16 BPE formats, R32_UINT for 32 BPE formats, and R32G32_UINT for 64 BPE formats. And the Channel Mask for surface write messages, and the Shader Channel Select fields 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



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).

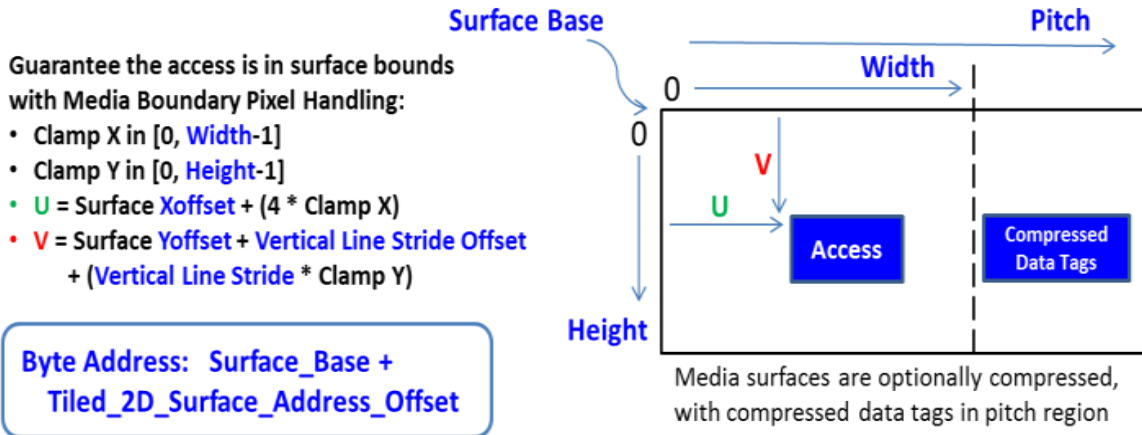
Upper left corner of the requested Block is specified by the signed (X, Y) coordinate.
 The block offset and block width are always aligned to size of pixel (BPE) in surface.

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



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])				u8	v6	u7	v5	u6	v4	u5	v3	u4	v2						
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])				u7	v7	u6	v6	u5	v5	u4	v4	v3	v2						
8	TileYS	8	8	u[13:8] + (v[13:8]*Pitch[17:8])				u7	v7	u6	v6	u5	v5	u4	v4	v3	v2						
	TileYF	6	6	u[13:6] + (v[13:6]*Pitch[17:6])				u6	v5	u4	v4	v3	v2	u5	u4	u3	u2						
all	TileY	5	7	u[13:7] + (v[13:5]*Pitch[17:7])				u6	u5	u4	v4	v3	v2	u5	u4	u3	u2	u1	u0				
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
<ul style="list-style-type: none"> • 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

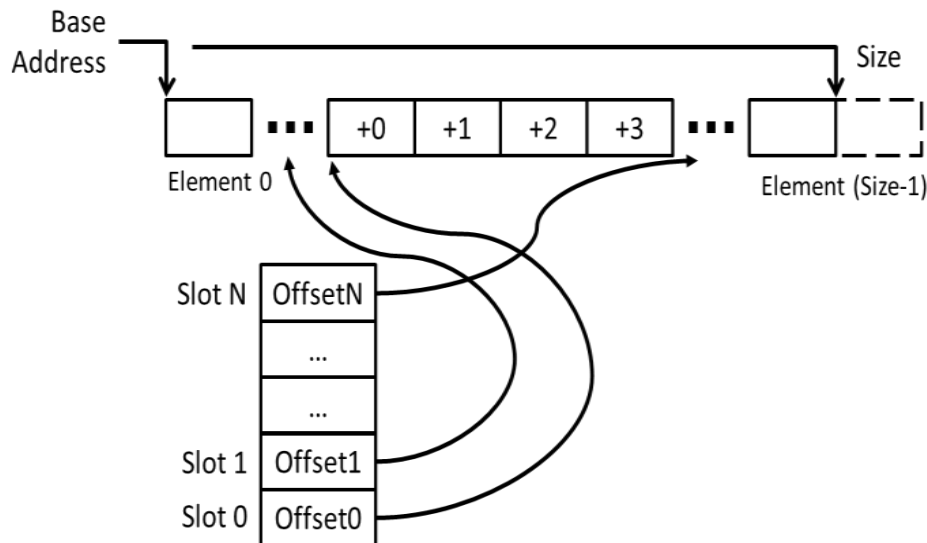
Surface Format Encoding (Hex)	Format Name	Bits Per Element(BPE)	Supported Compressible Media Surface Formats
0C0	B8G8R8A8_UNORM	32	<i>Not supported</i>
0C2	R10G10B10A2_UNORM	32	<i>Not supported</i>
0C4	R10G10B10A2_UINT	32	<i>Not supported</i>
0C7	R8G8B8A8_UNORM	32	<i>Not supported</i>
0C9	R8G8B8A8_SNORM	32	<i>Not supported</i>
0CA	R8G8B8A8_SINT	32	<i>Not supported</i>
0CB	R8G8B8A8_UINT	32	<i>Not supported</i>
0CC	R16G16_UNORM	32	<i>Not supported</i>
0CD	R16G16_SNORM	32	<i>Not supported</i>
0CE	R16G16_SINT	32	<i>Not supported</i>

Surface Format Encoding (Hex)	Format Name	Bits Per Element(BPE)	Supported Compressible Media Surface Formats
0CF	R16G16_UINT	32	<i>Not supported</i>
0D0	R16G16_FLOAT	32	<i>Not supported</i>
0D1	B10G10R10A2_UNORM	32	<i>Not supported</i>
0D3	R11G11B10_FLOAT	32	<i>Not supported</i>
0D5	R10G10B10_FLOAT_A2_UNORM	32	<i>Not supported</i>
0D6	R32_SINT	32	<i>Not supported</i>
0D7	R32_UINT	32	<i>Not supported</i>
0D8	R32_FLOAT	32	<i>Not supported</i>
100	B5G6R5_UNORM	16	<i>Not supported</i>
102	B5G5R5A1_UNORM	16	<i>Not supported</i>
104	B4G4R4A4_UNORM	16	<i>Not supported</i>
106	R8G8_UNORM	16	NV12 (UV 8bpe)
107	R8G8_SNORM	16	<i>Not supported</i>
108	R8G8_SINT	16	<i>Not supported</i>
109	R8G8_UINT	16	NV12 (UV 8bpe)
10A	R16_UNORM	16	Y16
10B	R16_SNORM	16	<i>Not supported</i>
10C	R16_SINT	16	<i>Not supported</i>
10D	R16_UINT	16	Y16
10E	R16_FLOAT	16	<i>Not supported</i>
113	A16_UNORM	16	Y16
11A	B5G5R5X1_UNORM	16	<i>Not supported</i>
140	R8_UNORM	8	NV12 (Y), Y8, IMC1 (YUV), IMC3 (YUV)
141	R8_SNORM	8	<i>Not supported</i>
142	R8_SINT	8	<i>Not supported</i>
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	<p>For SIMD16, the 16 bits of the execution mask control the 16 SIMD channels.</p> <p>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.</p>
SMBLK	SIMD8 or SIMD16 With Data Elements	<p>For SIMD16, the 16 bits of the execution mask control the 16 SIMD channels.</p> <p>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.</p> <p>If the data elements > 1, then the execution mask bits are reused for the additional data elements.</p>
SG	SIMD8 Slot Group	<p>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.</p>
BLK	Block	<p>The lower 8 bits of the execution mask control the first 8 DWords, and the upper 8 bits control the second 8 DWords.</p> <p>The DWords are accessed if the corresponding DWord's execution mask bit is set.</p> <p>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.</p>
BLKCM	Block with Channel Mode	<p>The lower 8 bits of the execution mask control the first 8 DWords, and the upper 8 bits control the second 8 DWords.</p> <p>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.</p>
Ignored	Not Used	<p>The execution mask bits are ignored by this message operation. The data is controlled by other parameters to this message.</p> <p>The execution mask bits must be non-zero for the message to be sent to the data port.</p>

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:

$$(Base + BaseOffset) \rightarrow DataType[Offset]$$

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	<p>Each Address Model has its own base address source, illustrated in the Surfaces Types section of this volume.</p> <p>The Base address is architecturally defined to always be aligned at a valid address boundary for the message.</p> <p>The notational convention in this chapter labels this calculated base address the Base.</p>
Buffer, BaseOffset	<p>The offsets from some messages are specified as byte offsets, and others are DWord (or other DataType) offsets.</p> <p>Buffer is shorthand for Buffer Base Address offset that is specified in A32 message headers.</p> <p>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.</p> <p>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.</p> <p>For most messages (except media block messages), the Buffer or BaseOffset value is always an unsigned (positive) value.</p>
B [], W [], DW [], QW [], OW [], HW []	<p>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.</p> <p>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.</p>
Offset{Slot}, U{Slot}, V{Slot}	<p>Offset{Slot} is shorthand for Offset0, Offset1, ... in a SIMD8 or SIMD16 address payload.</p> <p>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.</p> <p>For most messages (except media block messages), the Offset is always an unsigned (positive) value.</p>
Elem, Chan	<p>Elem represents the index into an array of consecutive data operands.</p> <p>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.</p>

Notation	Description
(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	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

Bounds Check	Address Model	Surface Type	Description
			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	Some A32 messages specify a Per Thread Scratch Space Size in their message header. When that message header is present and Local Thread Checking is enabled in GT Mode Register , the A32 message detects any address offsets that, directly or through the offset calculations, exceed the Per Thread Scratch Space Size. These A32 messages must additionally perform the GenState boundary check because the PTSS value is not guaranteed by trusted software to be within the GenState boundary limits.
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 the offset exceeds the allocated SLM memory size, the offset wraps around in the workgroup's allocated memory boundary and treated as an in-bound access.
Untyped2D	FLAT	BUFFER	If any byte(s) of the Untyped 2D block being accessed is outside of the underlying 2D Surface boundaries (as specified by the message header), those byte(s) will be considered as 'out-of-bound.' Hardware will return zeros for all reads from out-of-bound bytes. Writes to out-of-bound memory locations will be dropped.

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.

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	<p>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".</p> <p>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".</p>
Surface Type	Specifies the supported surface type. See the Surfaces Types section of this volume.
SIMD Slots	<p>In the first table, this summarizes all the SIMD slot configurations supported for this operation and address model.</p> <p>In the second table, this specifies the specific SIMD slot configuration supported by the specific message operation.</p>
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 message 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.
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.

Column Heading	Description
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	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
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	A32	BUFFER	1	1, 2, 4, 8	PTSS	BLKCM
HW Aligned Block R/W	HW	HW	A64	BUFFER	1	1, 2, 4, 8	Canonical	BLKCM
Surface R/W					2, 8, 16	1		
Scattered Untyped	DW	DW	BTS	BUFFER, NULL	8, 16	{Chan 1-4}	Surface	SM

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
Surface R/W	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 Atomic Float	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
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

Operation	Addr Align	Data Width	Address Model	Surface Type	SIMD Slots	Data Elements	Bounds Check	Execution Mask
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
Memory Fence	N/A	N/A	N/A	N/A	N/A	N/A	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

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	{Opt} MH_BTS_GO	MAP32B_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	BTS	BUFFER, NULL	SIMD8	{1, 2, 4}	MSD0W_BS	{Opt} MH_BTS_GO	MAP32B_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER, NULL	SIMD16	{1, 2, 4}	MSD0R_BS	{Opt} MH_BTS_GO	MAP32B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	BTS	BUFFER, NULL	SIMD16	{1, 2, 4}	MSD0W_BS	{Opt} MH_BTS_GO	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	{Opt} MH_A32_GO	MAP32B_SIMD8	{Forbidden}	{1, 1, 1} MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	{1, 2, 4}	MSD0W_BS	{Opt} MH_A32_GO	MAP32B_SIMD8	{1, 1, 1} MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	{1, 2, 4}	MSD0R_BS	{Opt} MH_A32_GO	MAP32B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{1, 2, 4}	MSD0W_BS	{Opt} MH_A32_GO	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/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	A64	BUFFER	SIMD16	{1, 2, 4}	MSD1R_A64_BS	{Forbidden}	MAP64B_SIMD16	{Forbidden}	{1, 1, 1} MDP_DW_SIMD16
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

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NU LL	SIMD8	1	MSD0R_DWS	{Opt} MH_BTS_G 0	MAP32B_SIMD 8	{Forbidden}	MDP_DW_SIMD 8
W	BTS	BUFFER,NU LL	SIMD8	1	MSD0W_DWS	{Opt} MH_BTS_G 0	MAP32B_SIMD 8	MDP_DW_SIMD 8	{Forbidden}
R	BTS	BUFFER,NU LL	SIMD1 6	1	MSD0R_DWS	{Opt} MH_BTS_G 0	MAP32B_SIMD 16	{Forbidden}	MDP_DW_SIMD 16
W	BTS	BUFFER,NU LL	SIMD1 6	1	MSD0W_DWS	{Opt} MH_BTS_G 0	MAP32B_SIMD 16	MDP_DW_SIMD 16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD0R_DWS	{Opt} MH_A32_G 0	MAP32B_SIMD 8	{Forbidden}	MDP_DW_SIMD 8
W	A32	BUFFER	SIMD8	1	MSD0W_DWS	{Opt} MH_A32_G 0	MAP32B_SIMD 8	MDP_DW_SIMD 8	{Forbidden}
R	A32	BUFFER	SIMD1 6	1	MSD0R_DWS	{Opt} MH_A32_G 0	MAP32B_SIMD 16	{Forbidden}	MDP_DW_SIMD 16
W	A32	BUFFER	SIMD1 6	1	MSD0W_DWS	{Opt} MH_A32_G 0	MAP32B_SIMD 16	MDP_DW_SIMD 16	{Forbidden}
R	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1R_A64_D WS	{Forbidden }	MAP64B_SIMD 8	{Forbidden}	{1, 2, 4} MDP_DW_SIMD 8
W	A64	BUFFER	SIMD8	{1, 2, 4}	MSD1W_A64_D WS	{Forbidden }	MAP64B_SIMD 8	{1, 2, 4} MDP_DW_SIMD 8	{Forbidden}
R	A64	BUFFER	SIMD1 6	{1, 2, 4}	MSD1R_A64_D WS	{Forbidden }	MAP64B_SIMD 16	{Forbidden}	{1, 2, 4} MDP_DW_SIMD 16
W	A64	BUFFER	SIMD1 6	{1, 2, 4}	MSD1W_A64_D WS	{Forbidden }	MAP64B_SIMD 16	{1, 2, 4} MDP_DW_SIMD 16	{Forbidden}
R	CC BTS	BUFFER,NU LL	SIMD8	1	MSD_CC_DWS	{Opt} MH_BTS_G 0	MAP32B_SIMD 8	{Forbidden}	MDP_DW_SIMD 8
R	CC BTS	BUFFER,NU LL	SIMD1 6	1	MSD_CC_DWS	{Opt} MH_BTS_G 0	MAP32B_SIMD 16	{Forbidden}	MDP_DW_SIMD 16

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 - 3FFFFFFh. 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	SLM	BUFFER	1	1, 2, 4, 8	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.	

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	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	A32	BUFFER	1	{1, 2, 4, 8}	MSD0R_OWB	MH_A32_GO	{Forbidden}	{Forbidden}	MDP_OW1L, MDP_OW1U, MDP_OW2, MDP_OW4
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,

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
									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 - 0FFFFFFh. 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.

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	A32	BUFFER	1	1, 2, 4, 8	(Base+Buffer)->HW[SB_Offset+Elem]	PTSS	BLKCM

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

Context:

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	A32	BUFFER	1	{1, 2, 4, 8}	MSD0R_SB	MH_A32_HWB	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
W	A32	BUFFER	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

For the MSD0R_SB and MSD0W_SB messages, the HWord offset into the A32 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) based from the A32 general state base address.

Scratch Block messages are always used inside a thread's local scratch space, and they are assumed to be non-coherent memory accesses (MDC_STATELESS A32_A64_NC).



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	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 DWord-aligned offset (Global Offset), and reads 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.

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}		MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
R	BTS	SCRATCH,NULL	1	{1, 2, 4, 8}		MH_SLM_GO	{Forbidden}	{Forbidden}	MDP_HW1, MDP_HW2, MDP_HW4, MDP_HW8
R	SLM	BUFFER	1	{1, 2, 4, 8}		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,	

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
								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	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.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NU LL	SIMD8	{Chan 1-4}	MSD1R_US	{Forbidde n}	MAP32b_USU_SIM D8	{Forbidden}	{Chan1-4} MDP_DW_SIMD 8
W	BTS	BUFFER,NU LL	SIMD8	{Chan 1-4}	MSD1W_US	{Forbidde n}	MAP32b_USU_SIM D8	{Chan1-4} MDP_DW_SIMD 8	{Forbidden}
R	BTS	BUFFER,NU LL	SIMD1 6	{Chan 1-4}	MSD1R_US	{Forbidde n}	MAP32b_USU_SIM D16	{Forbidden}	{Chan1-4} MDP_DW_SIMD 16
W	BTS	BUFFER,NU LL	SIMD1 6	{Chan 1-4}	MSD1W_US	{Forbidde n}	MAP32b_USU_SIM D16	{Chan1-4} MDP_DW_SIMD 16	{Forbidden}
R	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD1R_US	{Forbidde n}	MAP32b_USU_SIM D8	{Forbidden}	{Chan1-4} MDP_DW_SIMD 8
W	SLM	BUFFER	SIMD8	{Chan 1-4}	MSD1W_US	{Forbidde n}	MAP32b_USU_SIM D8	{Chan1-4} MDP_DW_SIMD 8	{Forbidden}
R	SLM	BUFFER	SIMD8	{Chan 1-4}		{Forbidde n}	MAP32b_USU_SIM D8	{Forbidden}	{Chan1-4} MDP_DW_SIMD 8
W	SLM	BUFFER	SIMD8	{Chan 1-4}		{Forbidde n}	MAP32b_USU_SIM D8	{Chan1-4} MDP_DW_SIMD 8	{Forbidden}
R	SLM	BUFFER	SIMD1 6	{Chan 1-4}	MSD1R_US	{Forbidde n}	MAP32b_USU_SIM D16	{Forbidden}	{Chan1-4} MDP_DW_SIMD 16
W	SLM	BUFFER	SIMD1 6	{Chan 1-4}	MSD1W_US	{Forbidde n}	MAP32b_USU_SIM D16	{Chan1-4} MDP_DW_SIMD 16	{Forbidden}
R	SLM	BUFFER	SIMD1 6	{Chan 1-4}	MSD1R_US	{Forbidde n}	MAP16b_USU_SIM D16	{Forbidden}	{Chan1-4} MDP_DW_SIMD 16
W	SLM	BUFFER	SIMD1 6	{Chan 1-4}	MSD1W_US	{Forbidde n}	MAP16b_USU_SIM D16	{Chan 1-4} MDP_DW_SIMD 16	{Forbidden}
R	SLM	BUFFER	SIMD1 6	{Chan 1-4}		{Forbidde n}	MAP32b_USU_SIM D16	{Forbidden}	{Chan1-4} MDP_DW_SIMD 16
W	SLM	BUFFER	SIMD1 6	{Chan 1-4}		{Forbidde n}	MAP32b_USU_SIM D16	{Chan1-4} MDP_DW_SIMD 16	{Forbidden}
R	A32	BUFFER	SIMD8	{Chan 1-4}	MSD1R_US	{Opt} MH1_A32	MAP32b_USU_SIM D8	{Forbidden}	{Chan1-4} MDP_DW_SIMD 8
W	A32	BUFFER	SIMD8	{Chan 1-4}	MSD1W_US	{Opt} MH1_A32	MAP32b_USU_SIM D8	{Chan1-4} MDP_DW_SIMD 8	{Forbidden}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	A32	BUFFER	SIMD16	{Chan 1-4}	MSD1R_US	{Opt} MH1_A32	MAP32b_USU_SIMD16	{Forbidden}	{Chan1-4} MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	{Chan 1-4}	MSD1W_US	{Opt} MH1_A32	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, NULL	SIMD 8	{Chan 1-4}	MSD1R_TS	{Forbidden}	MAP32b_TS_SIMD 8	{Forbidden}	{Chan1-4} MDP_DW_SIMD 8
W	BTS	1D, 2D, 3D, CUBE, BUFFER, NULL	SIMD 8	{Chan 1-4}	MSD1W_TS	{Forbidden}	MAP32b_TS_SIMD 8	{Chan1-4} MDP_DW_SIMD 8	{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 **CMPRW16B** 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.

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

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	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	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	{Opt} MH1_A32	MAP32b_USU_SIMD8	{Forbidden}	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	{Forbidden}	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAI	{Opt} MH1_A32	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	A32	BUFFER	SIMD8	1	MSD1R_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAI	{Opt} MH1_A32	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
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.

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
R	BTS	BUFFER,NUL L	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	BTS	BUFFER,NUL L	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NUL L	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	BTS	BUFFER,NUL L	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	BTS	BUFFER,NUL L	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	BUFFER,NUL L	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	BTS	BUFFER,NUL L	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	BTS	BUFFER,NUL L	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	BTS	BUFFER,NUL L	SIMD8	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	BTS	BUFFER,NUL L	SIMD8	1	MSD1W_DWAI	{Forbidden}	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	BTS	BUFFER,NUL L	SIMD16	1	MSD1R_DWAI	{Forbidden}	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	BTS	BUFFER,NUL L	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

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
			6			}			
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
W	SLM	BUFFER	SIMD16	1	MSD1W_DWAI	{Forbidden}	MAP16b_USU_SIMD16	MDP_AOP16_DW2	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	{Forbidden}	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	{Forbidden}	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	{Forbidden}	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAI	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAI	{Opt} MH1_A32	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

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
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.

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.

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

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_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	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_W_SIMD8	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAF	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_W_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAF	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_W_SIMD16	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAF	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_W_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_WAF	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_W2	MDP_W_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_WAF	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_W2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_WAF	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_AOP16_W2	MDP_W_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_WAF	{Opt} MH1_A32	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.

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

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
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	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_DW_SIMD8	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAF	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_DW_SIMD8	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAF	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_DW_SIMD16	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAF	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_DW_SIMD16	{Forbidden}
R	A32	BUFFER	SIMD8	1	MSD1R_DWAF	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_DW2	MDP_DW_SIMD8
W	A32	BUFFER	SIMD8	1	MSD1W_DWAF	{Opt} MH1_A32	MAP32b_USU_SIMD8	MDP_AOP8_DW2	{Forbidden}
R	A32	BUFFER	SIMD16	1	MSD1R_DWAF	{Opt} MH1_A32	MAP32b_USU_SIMD16	MDP_AOP16_DW2	MDP_DW_SIMD16
W	A32	BUFFER	SIMD16	1	MSD1W_DWAF	{Opt} MH1_A32	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	{Forbidden}	{Forbidden}	MDP_W_SIMD8	MDP_W_SIMD8
W	BTS	Any	SIMD8	1	MSD1W_WAC	{Forbidden}	{Forbidden}	MDP_W_SIMD8	{Forbidden}

Programming Note

Context: Word Atomic Counter Messages

- The unary atomic integer operations require a message header (so that the message data on the OBUS is not empty).
- 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.

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	{Forbidden}	{Forbidden}	MDP_DW_SIMD8	MDP_DW_SIMD8
W	BTS	Any	SIMD8	1	MSD1W_DWAC	{Forbidden}	{Forbidden}	MDP_DW_SIMD8	{Forbidden}

Programming Note

Context:

DWord Atomic Counter Messages

- The unary atomic integer operations require a message header (so that the message data on the OBUS is not empty).
- 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	1																								
	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	3																								
	4	40	39	38	37	36	35	34	33	20	19	18	17	10	9	5	4																								
	5	48	47	46	45	44	43	42	41	24	23	22	21	12	11	6	5																								
	6	56	55	54	53	52	51	50	49	28	27	26	25	14	13	7	6																								
	7	64	63	62	61	60	59	58	57	32	31	30	29	16	15	8	7																								

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 Read:

- For media block reads, the Block Width must be 32 or less.

Restrictions Media Block Write:

- For media block writes, both X Offset and Block Width must be DWord-aligned.

Programming Note	
Context:	Media Block Read/Write Messages
<ul style="list-style-type: none"> 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

		Width = 1-4								Width = 5-8								Width = 9-16								Width = 17-32							
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
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.

Memory 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	BTS	N/A	N/A	N/A	N/A	Ignored	Ignored
N/A	N/A	N/A	SLM	N/A	N/A	N/A	N/A	Ignored	Ignored

A memory fence message issued by a thread causes further messages issued by the thread to be blocked until all previous data port messages have completed, or the results can be globally observed from the point of view of other threads in the system. This includes both read and write messages.

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	BTS	N/A	N/A	N/A	MSD_MEMFENCE	MH_IGNORE	{Forbidden}	{Forbidden}	MDP_DW_SIMD8
N/A	SLM	N/A	N/A	N/A	MSD_MEMFENCE	MH_IGNORE	{Forbidden}	{Forbidden}	MDP_DW_SIMD8

The memory 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 data port messages are globally observable. The writeback register must be read before this thread sends another data port message.

A memory fence memory is typically performed prior to a Gateway Barrier message (which synchronizes threads in a group) so that when the barrier releases the threads, each thread is ensured its next data port message is ordered after the memory fence. See Read/Write Ordering in this volume for more details.

Global observability of SLM memory means the SLM memory is visible to all the threads in the thread group that have access to that memory. Global observability of the coherent L3 cache means that those portions of the L3 cache are observable both to graphics threads and to the rest of the system. Global observability of non-coherent L3 cache means those portions of the L3 cache are observable to graphics threads but not necessarily to the rest of the system.

The memory fence message descriptor also has controls to flush (invalidate) various caches. These controls are generally not used.

Programming Note	
Context:	Memory Fence Message
<ul style="list-style-type: none"> • The L3 has a few different partitioning schemes. In some cases, RW data, Constant data and/or texture data can be mixed in the same partition. If a flush is needed for any data type in the partition the entire partition is flushed. • When a cache flush control is specified to the L3 cache, the completion of the memory fence message does not mean that the cache flush has completed. When a thread needs the flush to be completed before continuing execution, the thread should issue a data port read or atomic operation (to any global address) after the cache flush. The cache flush will be completed when this global memory read or atomic operation completes. 	

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.

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 all channel accesses on the corresponding SIMD slots.

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 252 is used to reference bindless resource operation. When 252 is used, 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.

Data Port Bindless Surface Extended Message Descriptor

Entries 240-251 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
MSDOR_BS - Byte Scattered Read MSD
MSDOW_BS - Byte Scattered Write MSD
MSDOR_DWS - Dword Scattered Read MSD
MSDOW_DWS - Dword Scattered Write MSD

Block Read/Write Messages

Message Descriptor
MSDOR_OWB - Oword Block Read MSD
MSDOW_OWB - Oword Block Write MSD
MSDOR_OWAB - Oword Aligned Block Read MSD
MSDOR_HWB - Scratch Block Read MSD
MSDOW_HWB - Scratch Block Write MSD

Other Data Port Messages

Message Descriptor
MSD_MEMFENCE - Memory Fence 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

Message Descriptor
MSD_CC_DWS - Constant Cache Dword Scattered Read MSD
MSD_CC_OWUB - Constant Cache Oword Block Read MSD
MSD_CC_OWUB - Constant Cache Oword Unaligned Block Read MSD
MSD_SC_OWUB - Sampler Cache Oword Unaligned Block Read MSD
MSD_SC_MB - Sampler Cache Media Block Read MSD

Other Messages

Message Descriptor
MSD_RSI - Read Surface Info 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
MAP32b_USUV_SIMD8 - SIMD8 Untyped STRBUF Surface 32-Bit Address Payload
MAP32b_USUV_SIMD16 - SIMD16 Untyped STRBUF Surface 32-Bit Address Payload
MAP16b USUV SIMD16 - SIMD16 Untyped STRBUF 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_MSAА_SN - MSAА 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

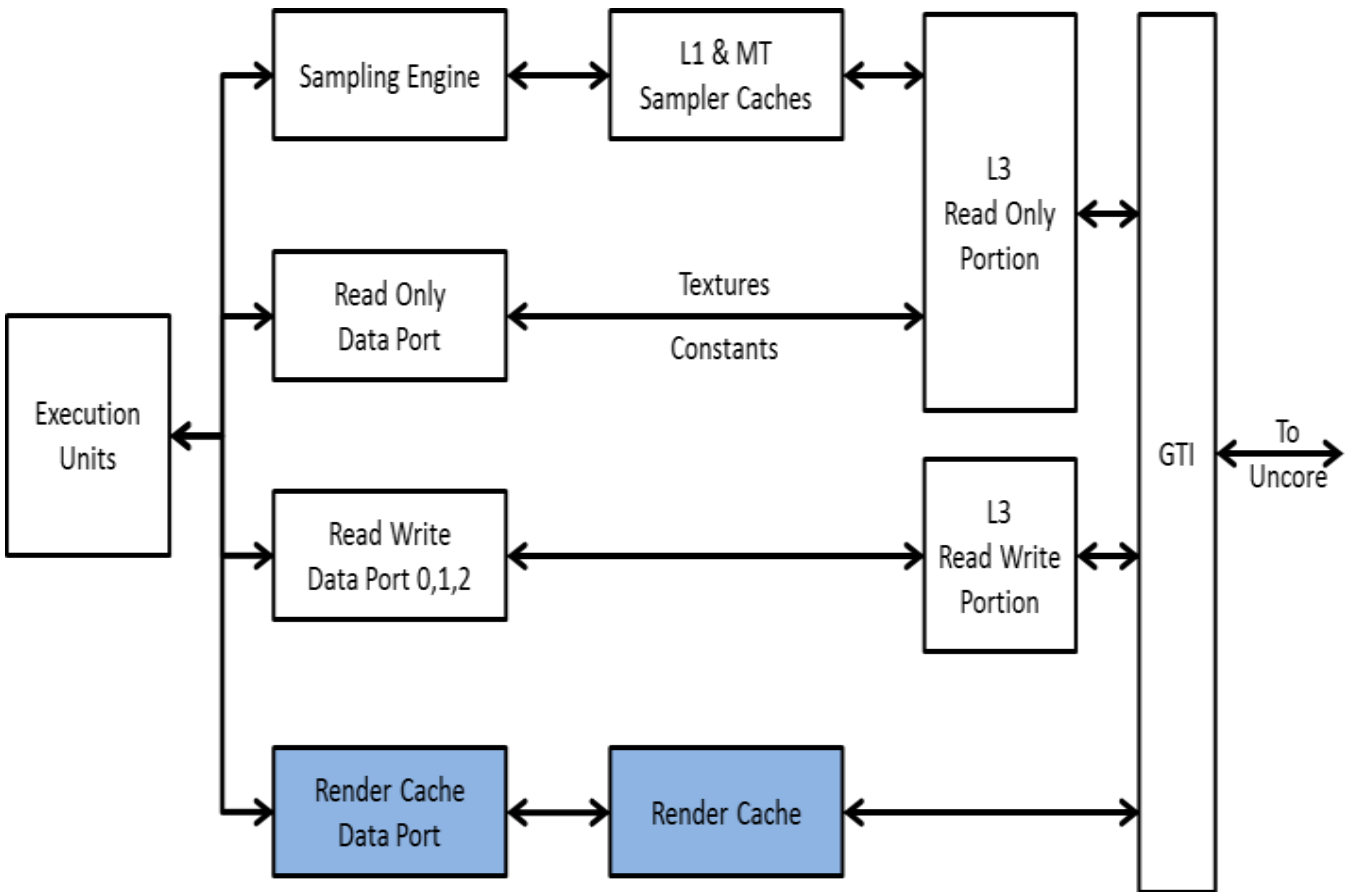
Control
MACD_URB_CMASK - URB Channel Mask Payload Control
MHC_URB_HANDLE - URB Handle Message Header Control

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.
- The pixel shader kernel requires no attributes, and delivers a value of 0xFFFFFFFF in all channels of the render target write message. The replicated color message should be used.
- **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 render target.
- After Render target fast clear, pipe-control with color cache write-flush must be issued before sending any DRAW commands on that render target.

Programming Note	
Context:	Workaround
<p>SW must try one of the following two options</p> <ol style="list-style-type: none"> 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
<p>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</p>	

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 with ClearColorConversion feature. This feature is enabled by setting **ClearColorConversionEnable**, it is default behavior. 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:

- SW must store clear color using MI_STORE_DATA_IMM.
- No further synchronization is required.
- SW must store clear color using two MI_ATOMIC commands using inline data mode, with **AtomicOpcode** set to MOV8B (0x24), **ReturnDataControl** and **CsStall** set to true on the last of those MI_ATOMIC, and false on the others.
- 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
- SW must store clear color using MI_STORE_DATA_IMM with **ForceWriteCompletionCheck** bit set.
- 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.

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
<ul style="list-style-type: none"> • 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 <i>ld2dms</i> and <i>sample2dms</i> 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 TileY, TileYs, or TileYf.

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

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
			is per each plane of the multi-sample PLANAR surface.

MSAA	Width of Clear Rect	Height of Clear Rect
2x	$\text{Ceil}(1/8 * \text{width})$	$\text{Ceil}(1/2 * \text{height})$
4X	$\text{Ceil}(1/8 * \text{width})$	$\text{Ceil}(1/2 * \text{height})$
8X	$\text{Ceil}(1/2 * \text{width})$	$\text{Ceil}(1/2 * \text{height})$
16X	width	$\text{Ceil}(1/2 * \text{height})$

- 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 TileY mode.
- Fast Color Clear:** 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

Restrictions									
Support is limited to tiled render targets.									
They are supported with MCS buffer layout with these alignments in the RT space: Horizontal Alignment = 128 and Vertical Alignment = 64.									
MCS and Lossless compression is supported for TiledY/TileYs/TileYf non-MSRTs only.									
Clear is supported only on the full RT; i.e., no partial clear or overlapping clears.									
For full resolve of a 3D surface compresses RT must be copied to a new surface									
Fast clear to 0 is not supported for MSAA > 1x									
The following table describes the RT alignment:									
<table border="1"> <thead> <tr> <th>TiledY RT CL</th> <th>Pixels</th> <th>Lines</th> </tr> </thead> <tbody> <tr> <td>bpp</td> <td></td> <td></td> </tr> <tr> <td>8</td> <td>32</td> <td>4</td> </tr> </tbody> </table>	TiledY RT CL	Pixels	Lines	bpp			8	32	4
TiledY RT CL	Pixels	Lines							
bpp									
8	32	4							

Restrictions		
16	16	4
32	8	4
64	4	4
128	2	4
TiledX RT CL		
bpp		
32	16	2
64	8	2
128	4	2

CCS buffer (non-MSRT) is supported only for RT formats 8bpp, 16bpp, 32bpp, 64bpp, and 128bpp.

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.

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

Tile-Y RT	Pixels	Lines
32	128	32
64	64	32
128	32	32

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 Tiled-YF RCC	Horizontal Scale Down Factor	Vertical Scale Down Factor
bpp		
8	128	32
16	128	16
32	64	16
64	64	8
128	32	8

CCS CL for Tiled-YS RCC	Horizontal Scale Down Factor	Vertical Scale Down Factor
bpp		
8	64	64
16	64	32
32	32	32
64	32	16
128	16	16

CCS CL for Tiled-Y RCC	Horizontal Scale Down Factor	Vertical Scale Down Factor
bpp		
8	256	16
16	128	16
32	64	16
64	32	16
128	16	16

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.

Programming Note

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). $CCS_Buffer_Size = Padded_Main_Surface_Size / 256$

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	<i>Not Supported</i>	Read/Write	All
R32G32B32A32_UINT	<i>Not Supported</i>	Read/Write	All
R32G32B32X32_FLOAT		Read/Write	
R16G16B16A16_UNORM	All	Read/Write	All
R16G16B16A16_SNORM	All	Read/Write	All
R16G16B16A16_SINT	<i>Not Supported</i>	Read/Write	All
R16G16B16A16_UINT	<i>Not Supported</i>	Read/Write	All
R16G16B16A16_FLOAT	All	Read/Write	All
R32G32_FLOAT	All	Read/Write	All
R32G32_SINT	<i>Not Supported</i>	Read/Write	All
R32G32_UINT	<i>Not Supported</i>	Read/Write	All
R16G16B16X16_FLOAT	All	Read/Write	All
B8G8R8A8_UNORM	All	Read/Write	All

Surface Format Name	Color Blend Support	Message Type Support	Lossless Compression Support
B8G8R8A8_UNORM_SRGB	All	Read/Write	
R10G10B10A2_UNORM	All	Read/Write	
R10G10B10A2_UINT	<i>Not Supported</i>	Read/Write	
R8G8B8A8_UNORM	All	Read/Write	All
R8G8B8A8_UNORM_SRGB	All	Read/Write	
R8G8B8A8_SNORM	All	Read/Write	All
R8G8B8A8_SINT	<i>Not Supported</i>	Read/Write	All
R8G8B8A8_UINT	<i>Not Supported</i>	Read/Write	All
R16G16_UNORM	All	Read/Write	All
R16G16_SNORM	All	Read/Write	All
R16G16_SINT	<i>Not Supported</i>	Read/Write	All
R16G16_UINT	<i>Not Supported</i>	Read/Write	All
R16G16_FLOAT	All	Read/Write	All
B10G10R10A2_UNORM	All	Read/Write	
B10G10R10A2_UNORM_SRGB	All	Read/Write	
R10G10B10_FLOAT_A2_UNORM		Read/Write	
R11G11B10_FLOAT	All	Read/Write	
R32_SINT	<i>Not Supported</i>	Read/Write	All
R32_UINT	<i>Not Supported</i>	Read/Write	All
R32_FLOAT	All	Read/Write	All
B8G8R8X8_UNORM	All	Read/Write	All
B8G8R8X8_UNORM_SRGB	All	Read/Write	
B5G6R5_UNORM	All	Read/Write	
B5G6R5_UNORM_SRGB	All	Read/Write	
B5G5R5A1_UNORM	All	Read/Write	
B5G5R5A1_UNORM_SRGB	All	Read/Write	
B4G4R4A4_UNORM	All	Read/Write	
B4G4R4A4_UNORM_SRGB	All	Read/Write	
R8G8_UNORM	All	Read/Write	
R8G8_SNORM	All	Read/Write	
R8G8_SINT	<i>Not Supported</i>	Read/Write	
R8G8_UINT	<i>Not Supported</i>	Read/Write	
R16_UNORM	All	Read/Write	
R16_SNORM	All	Read/Write	
R16_SINT	<i>Not Supported</i>	Read/Write	
R16_UINT	<i>Not Supported</i>	Read/Write	
R16_FLOAT	All	Read/Write	

Surface Format Name	Color Blend Support	Message Type Support	Lossless Compression Support
B5G5R5X1_UNORM	All	Read/Write	
B5G5R5X1_UNORM_SRGB	All	Read/Write	
A1B5G5R5_UNORM	All	Read/Write	
A4B4G4R4_UNORM	All	Read/Write	
R8_UNORM	All	Read/Write	
R8_SNORM	All	Read/Write	
R8_SINT	<i>Not Supported</i>	Read/Write	
R8_UINT	<i>Not Supported</i>	Read/Write	
A8_UNORM	All	Read/Write	
YCRCB_NORMAL	<i>Not Supported</i>	Write	<i>Not Supported</i>
YCRCB_SWAPUVY	<i>Not Supported</i>	Write	<i>Not Supported</i>
YCRCB_SWAPUV	<i>Not Supported</i>	Write	<i>Not Supported</i>
YCRCB_SWAPY	<i>Not Supported</i>	Write	<i>Not Supported</i>

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
<ul style="list-style-type: none"> • 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 <i>send</i> 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 operations.
Writeback Payload	Returns result data for read operations 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}
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}

R/W	Address Model	Surface Type	SIMD Slots	Data Elements	Message Specific Descriptor	Message Header	Address Payload	Source Payload	Writeback Payload
		CUBE, BUFFER							
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}
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}

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_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}
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}

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_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	1/0s0A	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 render target message 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.	

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 must send messages in the following sequence for each RT: SIMD8 dual source RTW message (low); SIMD8 dual source RTW message (high); SIMD16 single src RTW message with second color.

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 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 Interpolator

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 <i>Pixel Interpolator</i> 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.

Bits	Description								
14	<p>Interpolation Mode. Specifies which interpolation mode is used.</p> <p>Format = U1</p> <p>0: Perspective Interpolation</p> <p>1: Linear Interpolation</p> <table border="1" data-bbox="220 485 1494 533"> <thead> <tr> <th colspan="2">Programming Note</th> </tr> </thead> <tbody> <tr> <td colspan="2">This field cannot be set to "Linear Interpolation" unless Non-Perspective Barycentric Enable in 3DSTATE_CLIP is enabled.</td> </tr> </tbody> </table> <table border="1" data-bbox="220 674 1221 722"> <thead> <tr> <th colspan="2">Programming Note</th> </tr> </thead> <tbody> <tr> <td>Context:</td> <td>Message Descriptor</td> </tr> </tbody> </table> <p>This field is ignored when Message Type is set to Coarse To Pixel Mapping message.</p>	Programming Note		This field cannot be set to "Linear Interpolation" unless Non-Perspective Barycentric Enable in 3DSTATE_CLIP is enabled.		Programming Note		Context:	Message Descriptor
Programming Note									
This field cannot be set to "Linear Interpolation" unless Non-Perspective Barycentric Enable in 3DSTATE_CLIP is enabled.									
Programming Note									
Context:	Message Descriptor								
13:12	<p>Message Type. Specifies the type of message being sent when pixel-rate evaluation is requested.</p> <p>Format = U2</p> <p>0: Per Message Offset (eval_snapped with immediate offset)</p> <p>1: Sample Position Offset (eval_sindex)</p> <p>2: Centroid Position Offset (eval_centroid)</p> <p>3: Per Slot Offset (eval_snapped with register offset)</p> <p>Message Type. Specifies the type of message being sent when coarse-rate evaluation is requested.</p> <p>Format = U2</p> <p>0: Coarse to Pixel Mapping Message (internal message)</p> <p>1: Reserved</p> <p>2: Coarse Centroid Position (eval_centroid)</p> <p>3: Per Slot Coarse Pixel Offset (eval_snapped with register offset)</p> <table border="1" data-bbox="220 1486 1494 1535"> <thead> <tr> <th colspan="2">Programming Note</th> </tr> </thead> <tbody> <tr> <td>Context:</td> <td>Message Descriptor</td> </tr> </tbody> </table> <p>When Message Type is Sample Position, requesting an attribute at sample index beyond the range defined by the Forced Sample Count is illegal.</p>	Programming Note		Context:	Message Descriptor				
Programming Note									
Context:	Message Descriptor								
11	<p>Note: For shaders dispatched at coarse rate, this bit has a different meaning and is part of the Pixel Shader Phase field.</p> <p>Slot Group Select. This field selects whether slots 15:0 or slots 31:16 are used for bypassed data.</p> <p>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</p>								

Bits	Description						
	<p>each message based on which slots are currently being processed.</p> <p>0: SLOTGRP_LO: Choose bypassed data for slots 15:0.</p> <p>1: SLOTGRP_HI: Choose bypassed data for slots 31:16.</p> <table border="1" data-bbox="220 422 1495 625"> <thead> <tr> <th colspan="2" data-bbox="220 422 1495 464">Programming Note</th> </tr> </thead> <tbody> <tr> <td data-bbox="220 464 646 506">Context:</td> <td data-bbox="646 464 1495 506">Message Descriptor</td> </tr> <tr> <td colspan="2" data-bbox="220 506 1495 625">This field must be set to SLOTGRP_LO for SIMD8 messages. SIMD8 messages always use bypassed data for slots 7:0.</td> </tr> </tbody> </table>	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.	
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	<p>For shaders not dispatched at coarse rate, bit 11 is the Slot Group Select bit and bits 10:8 are ignored.</p> <p>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.</p> <p>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.</p> <p>Format: U4</p> <p>Range = [0..15]</p> <p>Values other than 0 are valid only for shaders dispatched at coarse shading rate.</p>						
7:0	<p>Message Specific Control. Refer to the sections below for the definition of these bits based on Message Type.</p>						

"Per Message Offset" Message Descriptor

Bit	Description
7:4	<p>Per Message Y Pixel Offset</p> <p>Specifies the Y Pixel Offset that applies to all slots.</p> <p>Format = S4 2's complement representing units of 1/16 pixel.</p> <p>Range = [-8/16, +7/16]</p>
3:0	<p>Per Message X Pixel Offset</p> <p>Specifies the X Pixel Offset that applies to all slots.</p> <p>Format = S4 2's complement representing units of 1/16 pixel.</p> <p>Range = [-8/16, +7/16]</p>

"Sample Position Offset" Message Descriptor

Bits	Description		
7:4	<p>Sample Index</p> <p>Specifies the sample index that applies to all slots.</p> <p>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.</p> <p>Format = U4</p> <table border="1"> <tr> <td>Range</td> </tr> <tr> <td>[0, 15]</td> </tr> </table>	Range	[0, 15]
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	<p>Coarse to Pixel Mapping Phase Enable bits</p> <p>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.</p> <p>bit 5: Output Pixel Coverage Mask Valid in payload</p> <p>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:</p> <p>bit 4: Perspective Pixel Center Location Barycentric</p> <p>bit 3: Perspective Pixel Centroid Location Barycentric</p> <p>bit 2: Linear Pixel Center Location Barycentric</p> <p>bit 1: Linear Pixel Centroid Location Barycentric</p> <p>bit 0: Pixel Input Coverage Mask</p>

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. Format = S4 2's complement representing units of 1/16 pixel. The upper 28 bits are ignored. Range = [-8/16, +7/16]
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
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

DWord	Bit	Description
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.

DWord	Bit	Description
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 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

DWord	Bit	Description
		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
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

DWord	Bit	Description
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

DWord	Bit	Description
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
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	MSSAA rate (multisample count) Format: U4 [1..16] This field specifies MSSAA sampling rate (required for PS+S monolithic shader).
	15:4	Reserved (MBZ)

	3:0	<p>Next Pixel Shader Phase</p> <p>Format = U4 [0..15]</p> <p>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.</p> <p>The intended usage scenario is</p> <ol style="list-style-type: none"> 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	<p>Perspective Pixel Location Barycentric[1] for Slot 7</p> <p>This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.</p> <p>Format = IEEE_Float</p>
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	<p>Perspective Pixel Location Barycentric[1] for Slot 7</p> <p>This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.</p> <p>Format = IEEE_Float</p>
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		<p>Pixel Shader Input Coverage Mask, for slots 7:0</p> <p>This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.</p> <p>Format = U32</p>
W19		<p>Pixel Shader Input Coverage Mask, for slots 15:8</p> <p>Format = U32</p> <p>This phase is included only if the corresponding bit in Coarse to Pixel Mapping message specific control is enabled.</p>

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

This section describes the Message Gateway messages.

Message Descriptor

The following message descriptor applies to all messages supported by Message Gateway.

Bits	Description																		
19	Header Present. This bit must be 0 for all Message Gateway messages.																		
18:17	Ignored.																		
16:3	Reserved: MBZ																		
2:0	<p>SubFuncID. Identify the supported sub-functions by Message Gateway. Encodings are:</p> <table border="1"> <thead> <tr> <th>Value</th> <th>Name</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>001b</td> <td>SignalEvent</td> <td>Sends an event to all threads that are monitoring that Event ID</td> </tr> <tr> <td>010b</td> <td>MonitorEvent</td> <td>Gateway will remember if this Event ID occurs</td> </tr> <tr> <td>011b</td> <td>MonitorNoEvent</td> <td>Gateway will no longer record any Events for sending thread.</td> </tr> <tr> <td>100b</td> <td>BarrierMsg</td> <td>Record an additional thread reaching the barrier</td> </tr> <tr> <td>110b</td> <td>WaitForEvent</td> <td>Send a writeback if Event ID occurred after MonitorEvent</td> </tr> </tbody> </table> <p>All other encodings Reserved.</p>	Value	Name	Description	001b	SignalEvent	Sends an event to all threads that are monitoring that Event ID	010b	MonitorEvent	Gateway will remember if this Event ID occurs	011b	MonitorNoEvent	Gateway will no longer record any Events for sending thread.	100b	BarrierMsg	Record an additional thread reaching the barrier	110b	WaitForEvent	Send a writeback if Event ID occurred after MonitorEvent
Value	Name	Description																	
001b	SignalEvent	Sends an event to all threads that are monitoring that Event ID																	
010b	MonitorEvent	Gateway will remember if this Event ID occurs																	
011b	MonitorNoEvent	Gateway will no longer record any Events for sending thread.																	
100b	BarrierMsg	Record an additional thread reaching the barrier																	
110b	WaitForEvent	Send a writeback if Event ID occurred after MonitorEvent																	

BarrierMsg Message

The BarrierMsg message gives the ability for multiple threads to synchronize their progress. This is useful when there are data shared between threads. The message consists of a single 256-bit message payload.

Upon receiving one such message, Message Gateway increments the Barrier counter and marks the Barrier requester thread. There is no immediate response from the Message Gateway when the incremented counter is not equal to the terminating thread count. When the incremented counter value does equal the **Barrier Thread Count**, Message Gateway sends a response back to all the Barrier requester threads and resets the Barrier count to zero.

Programming Note	
Context:	BarrierMsg Message
<p>The Message Gateway assumes that the barrier ID sent in barrier message payload is valid and was allocated by TSG. In the event of a programming error specifying an invalid barrier ID, the results are undefined, and may cause the Message Gateway to stop responding to barrier messages for any thread that it services.</p>	

Message Payload

DWord	Bits	Description
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Predicate Mask. This field has a bit set per SIMD channel that passes the predicate. For SIMD8 and SIMD16 the rest of the bits must be 0. This field is ignored for non-predicated barriers.
M0.2	31	Ignored
	30:24	BarrierID. This field indicates which barrier state is updated. Format: U7 Range = [0, 63]
	23	Predicate Mask Enable. This bit indicates that the barrier is a predicated barrier and the SIMD channels passing the predicate should be summed. All threads sending this message to the same barrier should have an identical value for this field, and must specify a response length of 1 for the predicate sum response. Note that Global Barriers must not have the Predicate Mask Enable bit set. Programming Note: This control is intended only for GPGPU or Media threads. This control must not be set if the barrier is for a Hull Shader thread. Format: Enable
	22:16	Ignored
	15	Barrier Count Enable. Allows the message to reprogram the terminating barrier count. If set, the stored value of the terminating barrier count is set to the value of Barrier Count field (below), and used for this barrier operation. If clear, the stored value of the terminating barrier count is not modified and the stored value is used for this barrier operation. Programming Note: This control is intended only for Hull Shader threads. This control must not be set if the barrier is allocated by a GPGPU or Media thread. Format: Enable
	14:8	Barrier Count. If Barrier Count Enable is set, this field specifies the terminating barrier count. Otherwise this field is ignored. All threads that belong to a single barrier must deliver the same value for this field for a particular barrier iteration. Format: U7

DWord	Bits	Description
	7:0	Ignored
M0.1	31:0	Ignored
M0.0	31:4	Ignored

Writeback Message to Requester Thread

The writeback message is sent only if the **AckReq** bit in the message descriptor is set.

Programming Note	
Context:	Writeback Message to Requester Thread
AckReq must not be used with predicate barriers, because the predicate barrier response is sent to the GRF writeback location instead of the Ack.	

DWord	Bits	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:20	Reserved
	19:16	Shared Function ID. Contains the message gateway's shared function ID.
	15:3	Reserved
	2:0	Error Code 000: <i>Successful</i> . No Error (Normal). Other encodings are reserved.

Broadcast Writeback Message

Description
When the count for a Barrier reaches Barrier.Count, the Message Gateway sends the notification bit N0 to each EU/Thread that reached the barrier. A Barrier Return Byte is not sent.
This response message is sent back only if the Gateway Barrier Message specifies that this is a predicated barrier. This response is written to the GRF writeback location, and the response length specified in the send message to the EU must be 1.

DWord	Bits	Description
W0.7:1		Reserved (not overwritten)
W0.0	31:16	Reserved (not overwritten)
	15:0	Predicated Barrier Mask Sum. This field is a sum of the predicate mask bits sent by each thread. This field (and the DW containing it) is not written if the barrier is not marked as a predicated barrier. The kernel should compare this field to 0 for the predicated OR function and compare it to the workgroup size for the predicated AND function.



MonitorEvent

The MonitorEvent message indicates that the Gateway should remember any events of the specified EventID for the messaging thread. No writeback message is sent until a WaitForEvent message arrives, however. Only one event can be monitored at a time per thread, so a new MonitorEvent message will replace the event being monitored by the sending thread with the new EventID. An event recorded for the thread by a previous MonitorEvent is cleared when a new MonitorEvent comes in.

Message Payload

DWord	Bits	Description
M0.7	31:0	Ignored
M0.6	31:0	Ignored
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:24	Ignored
	23:0	EventID - Indicates the ID of the event to be monitored.

WaitForEvent

The WaitForEvent message causes the Gateway to send a writeback message to the thread when a monitored event occurs, or to send the writeback immediately if the event occurred before the WaitForEvent and after a MonitorEvent. A timeout value is supplied to cause the Gateway to send the writeback after the timeout occurs. Timeout always happens during pre-emption.

Programming Note

The return of a WaitForEvent writeback message does not mean that the monitored event occurred. A timeout or preemption may have occurred, or the signal could have been sent by a previous signaling phase, or possibly even by another program that is using the same event number.

Therefore, each monitor event should be used in combination with a global memory location that is written and read with atomic messages. That memory location records the signal, and the monitor event becomes a way to know when to poll the global memory location to see if the event has occurred.

Message Payload

DWord	Bits	Description
M0.7	31:0	Ignored
M0.6	31:0	Ignored
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:10	Ignored
	9:0	Timeout- The amount of time GW should wait before sending a writeback message. This value is in terms of 1024 clocks, so with a 1Ghz clock it would be approximately in terms of uS. 0 and 1 are illegal values, since the actual timeout time can be short by up to 1 increment of the timeout value.

Writeback Message to Requester Thread

DWord	Bit	Description
R0.7	31:0	Ignored
R0.6	31:0	Ignored
R0.5	31:0	Ignored
R0.4	31:0	Ignored
R0.3	31:0	Ignored
R0.2	31:0	Ignored
R0.1	31:0	Ignored
R0.0	31:1	Ignored
	0	Event Occured - This bit is 1 if the event occurred, or 0 if a timeout occurred.

Programming Note

Timing race conditions exist between one thread signaling an event, and another thread waiting for the event. It is possible that the WaitForEvent times out even when a signal has been sent. It is also possible that a WaitForEvent receives an EventOccurred from a prior signaling event that wasn't cleared out.

The correct programming model for Monitors uses atomic operations on a memory location shared by the threads for synchronization and monitors to reduce polling overhead on the memory location. The signaling thread first specifies the event with an atomic operation and then signals the event. The receiving thread waits for the event (which might be correct, might be a timeout or might be an old signal), and then verifies the event by checking the shared memory location.



SignalEvent

This message indicates to the rest of the system that the specified event has occurred. This event will be recorded for all threads monitoring it, and messages will be sent to all threads with an outstanding WaitForEvent.

Message Payload

DWord	Bits	Description
M0.7	31:0	Ignored
M0.6	31:0	Ignored
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:24	Ignored
	23:0	EventID - Indicates the ID of the event to be signalled.

MonitorNoEvent

This message indicates that the thread no longer wishes to monitor any events. The EOT message at the end of the thread does the same thing, so this message is optional for most usages. It is not needed when switching between different events - just a new MonitorEvent with the new Event ID suffices.

Message Payload

DWord	Bits	Description
M0.7	31:0	Ignored
M0.6	31:0	Ignored
M0.5	31:0	Ignored
M0.4	31:0	Ignored
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:0	Ignored
M0.0	31:0	Ignored

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.

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
<pre>extern int semaphore = 0; // do work, produce results ... atomic_store(&semaphore, 1); signal_event(&semaphore);</pre>	<pre>extern int semaphore; monitor_event(&semaphore); while ((atomic_cmpxchg(&semaphore, 1, 0) == 0) { wait_event(); } monitor_no_event(); // consume other thread's results ...</pre>

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.